33rd International Symposium on Distributed Computing

DISC 2019, October 14–18, 2019, Budapest, Hungary

Edited by

Jukka Suomela

Editors

Jukka Suomela
Aalto University, Finland
jukka.suomela@aalto.fi

ACM Classification 2012
Software and its engineering → Distributed systems organizing principles; Computing methodologies → Distributed computing methodologies; Computing methodologies → Concurrent computing methodologies; Hardware → Fault tolerance; Networks; Information systems → Data structures; Theory of computation; Theory of computation → Models of computation; Theory of computation → Design and analysis of algorithms

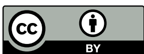
ISBN 978-3-95977-126-9

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-126-9.

Publication date
October, 2019

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at https://portal.dnb.de.

License
This work is licensed under a Creative Commons Attribution 3.0 Unported license (CC-BY 3.0):
https://creativecommons.org/licenses/by/3.0/legalcode.



In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work under the following conditions, without impairing or restricting the authors' moral rights:

- Attribution: The work must be attributed to its authors.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/LIPIcs.DISC.2019.0

ISBN 978-3-95977-126-9 **ISSN 1868-8969** **https://www.dagstuhl.de/lipics**

LIPIcs – Leibniz International Proceedings in Informatics

LIPIcs is a series of high-quality conference proceedings across all fields in informatics. LIPIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

Editorial Board

- Luca Aceto (*Chair*, Gran Sasso Science Institute and Reykjavik University)
- Christel Baier (TU Dresden)
- Mikolaj Bojanczyk (University of Warsaw)
- Roberto Di Cosmo (INRIA and University Paris Diderot)
- Javier Esparza (TU München)
- Meena Mahajan (Institute of Mathematical Sciences)
- Dieter van Melkebeek (University of Wisconsin-Madison)
- Anca Muscholl (University Bordeaux)
- Luke Ong (University of Oxford)
- Catuscia Palamidessi (INRIA)
- Thomas Schwentick (TU Dortmund)
- Raimund Seidel (Saarland University and Schloss Dagstuhl – Leibniz-Zentrum für Informatik)

ISSN 1868-8969

https://www.dagstuhl.de/lipics

Contents

Preface
Jukka Suomela 0:ix–0:x

Symposium Organization
.... 0:xi–0:xiv

2019 Edsger W. Dijkstra Prize in Distributed Computing
.... 0:xv

2019 Principles of Distributed Computing Doctoral Dissertation Award
.... 0:xvii

Regular Papers

Consensus with Max Registers
James Aspnes and He Yang Er 1:1–1:9

Putting Strong Linearizability in Context: Preserving Hyperproperties in Programsthat Use Concurrent Objects
Hagit Attiya and Constantin Enea 2:1–2:17

Long-Lived Counters with Polylogarithmic Amortized Step Complexity
Mirza Ahad Baig, Danny Hendler, Alessia Milani, and Corentin Travers 3:1–3:16

Distributed Algorithms for Low Stretch Spanning Trees
Ruben Becker, Yuval Emek, Mohsen Ghaffari, and Christoph Lenzen 4:1–4:14

Optimal Distributed Covering Algorithms
Ran Ben-Basat, Guy Even, Ken-ichi Kawarabayashi, and Gregory Schwartzman .. 5:1–5:15

Parameterized Distributed Algorithms
Ran Ben-Basat, Ken-ichi Kawarabayashi, and Gregory Schwartzman 6:1–6:16

Message Reduction in the LOCAL Model Is a Free Lunch
Shimon Bitton, Yuval Emek, Taisuke Izumi, and Shay Kutten 7:1–7:15

On the Computational Power of Radio Channels
Mark Braverman, Gillat Kol, Rotem Oshman, and Avishay Tal 8:1–8:17

Space-Optimal Naming in Population Protocols
Janna Burman, Joffroy Beauquier, and Devan Sohier 9:1–9:16

Erasure Correction for Noisy Radio Networks
Keren Censor-Hillel, Bernhard Haeupler, D. Ellis Hershkowitz, and Goran Zuzic . 10:1–10:17

Reachability and Shortest Paths in the Broadcast CONGEST Model
Shiri Chechik and Doron Mukhtar 11:1–11:13

On the Round Complexity of Randomized Byzantine Agreement
Ran Cohen, Iftach Haitner, Nikolaos Makriyannis, Matan Orland, and Alex Samorodnitsky 12:1–12:17

33rd International Symposium on Distributed Computing (DISC 2019).
Editor: Jukka Suomela

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Trade-Offs in Distributed Interactive Proofs
Pierluigi Crescenzi, Pierre Fraigniaud, and Ami Paz 13:1–13:17

The Capacity of Smartphone Peer-To-Peer Networks
Michael Dinitz, Magnús M. Halldórsson, Calvin Newport, and Alex Weaver 14:1–14:17

Sublinear-Time Distributed Algorithms for Detecting Small Cliques and Even Cycles
Talya Eden, Nimrod Fiat, Orr Fischer, Fabian Kuhn, and Rotem Oshman 15:1–15:16

A Distributed Algorithm for Directed Minimum-Weight Spanning Tree
Orr Fischer and Rotem Oshman 16:1–16:16

Stable Memoryless Queuing under Contention
Paweł Garncarek, Tomasz Jurdziński, and Dariusz R. Kowalski 17:1–17:16

Improved Network Decompositions Using Small Messages with Applications on MIS, Neighborhood Covers, and Beyond
Mohsen Ghaffari and Julian Portmann 18:1–18:16

On Bioelectric Algorithms
Seth Gilbert, James Maguire, and Calvin Newport 19:1–19:17

Parallel Finger Search Structures
Seth Gilbert and Wei Quan Lim 20:1–20:18

Wait-Free Solvability of Equality Negation Tasks
Éric Goubault, Marijana Lazić, Jérémy Ledent, and Sergio Rajsbaum 21:1–21:16

Scalable Byzantine Reliable Broadcast
Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi 22:1–22:16

Fast Distributed Algorithms for LP-Type Problems of Low Dimension
Kristian Hinnenthal, Christian Scheideler, and Martijn Struijs 23:1–23:16

Privatization-Safe Transactional Memories
Artem Khyzha, Hagit Attiya, and Alexey Gotsman 24:1–24:17

Low-Congestion Shortcut and Graph Parameters
Naoki Kitamura, Hirotaka Kitagawa, Yota Otachi, and Taisuke Izumi 25:1–25:17

The Complexity of Symmetry Breaking in Massive Graphs
Christian Konrad, Sriram V. Pemmaraju, Talal Riaz, and Peter Robinson 26:1–26:18

Stellar Consensus by Instantiation
Giuliano Losa, Eli Gafni, and David Mazières 27:1–27:15

A Scalable, Portable, and Memory-Efficient Lock-Free FIFO Queue
Ruslan Nikolaev 28:1–28:16

Byzantine Approximate Agreement on Graphs
Thomas Nowak and Joel Rybicki 29:1–29:17

Small Cuts and Connectivity Certificates: A Fault Tolerant Approach
Merav Parter 30:1–30:16

Monotonically Relaxing Concurrent Data-Structure Semantics for Increasing Performance: An Efficient 2D Design Framework
Adones Rukundo, Aras Atalar, and Philippas Tsigas 31:1–31:15

Phase Transitions of Best-of-Two and Best-of-Three on Stochastic Block Models
Nobutaka Shimizu and Takeharu Shiraga 32:1–32:17

Distributed Data Summarization in Well-Connected Networks
Hsin-Hao Su and Hoa T. Vu 33:1–33:16

Polynomial-Time Fence Insertion for Structured Programs
Mohammad Taheri, Arash Pourdamghani, and Mohsen Lesani 34:1–34:17

Brief Announcements

Brief Announcement: On Self-Adjusting Skip List Networks
Chen Avin, Iosif Salem, and Stefan Schmid 35:1–35:3

Brief Announcement: Streaming and Massively Parallel Algorithms for Edge Coloring
Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi Hajiaghayi, Marina Knittel, and Hamed Saleh 36:1–36:3

Brief Announcement: Memory Lower Bounds for Self-Stabilization
Lélia Blin, Laurent Feuilloley, and Gabriel Le Bouder 37:1–37:3

Brief Announcement: Wait-Free Universality of Consensus in the Infinite Arrival Model
Grégoire Bonin, Achour Mostéfaoui, and Matthieu Perrin 38:1–38:3

Brief Announcement: Asymmetric Distributed Trust
Christian Cachin and Björn Tackmann 39:1–39:3

Brief Announcement: Implementing Byzantine Tolerant Distributed Ledger Objects
Vicent Cholvi, Antonio Fernández Anta, Chryssis Georgiou, and Nicolas Nicolaou 40:1–40:3

Brief Announcement: Model Checking Rendezvous Algorithms for Robots with Lights in Euclidean Space
Xavier Défago, Adam Heriban, Sébastien Tixeuil, and Koichi Wada 41:1–41:3

Brief Announcement: Massively Parallel Approximate Distance Sketches
Michael Dinitz and Yasamin Nazari 42:1–42:3

Brief Announcement: Neighborhood Mutual Remainder and Its Self-Stabilizing Implementation of Look-Compute-Move Robots
Shlomi Dolev, Sayaka Kamei, Yoshiaki Katayama, Fukuhito Ooshita, and Koichi Wada 43:1–43:3

Brief Announcement: Revisiting Consensus Protocols through Wait-Free Parallelization
Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi 44:1–44:3

Brief Announcement: The Fault-Tolerant Cluster-Sending Problem
Jelle Hellings and Mohammad Sadoghi 45:1–45:3

Brief Announcement: On the Correctness of Transaction Processing with External Dependency
Masoomeh Javidi Kishi, Ahmed Hassan, and Roberto Palmieri 46:1–46:3

Brief Announcement: Towards Byzantine Broadcast in Generalized Communication and Adversarial Models
Chen-Da Liu-Zhang, Varun Maram, and Ueli Maurer 47:1–47:3

Brief Announcement: Integrating Temporal Information to Spatial Information in a Neural Circuit
Nancy Lynch and Mien Brabeeba Wang 48:1–48:3

Brief Announcement: Faster Asynchronous MST and Low Diameter Tree Construction with Sublinear Communication
Ali Mashreghi and Valerie King 49:1–49:3

Preface

DISC, the International Symposium on Distributed Computing, is an international forum on the theory, design, analysis, implementation and application of distributed systems and networks. DISC is organized in cooperation with the European Association for Theoretical Computer Science (EATCS).

This volume contains the papers presented at DISC 2019, the 33rd International Symposium on Distributed Computing, held on October 14–18, 2019 in Budapest, Hungary. It also includes the citations for two awards jointly sponsored by DISC and the ACM Symposium on Principles of Distributed Computing (PODC):

- The *2019 Edsger W. Dijkstra Prize in Distributed Computing*, presented at DISC 2019 to Alessandro Panconesi and Aravind Srinivasan for their paper "Randomized Distributed Edge Coloring via an Extension of the Chernoff–Hoeffding Bounds", SIAM Journal on Computing, volume 26, number 2, 1997, pages 350–368.
- The *2019 Principles of Distributed Computing Doctoral Dissertation Award*, presented at PODC 2019 to Sepehr Assadi for his dissertation "Combinatorial Optimization on Massive Datasets: Streaming, Distributed, and Massively Parallel Computation", written under the supervision of Prof. Sanjeev Khanna at the University of Pennsylvania.

In response to the call for papers, this year we received 145 regular submissions (four of them withdrawn after submission), and 13 brief submissions. All submissions were evaluated by at least three reviewers; we had a program committee with 38 members, and the committee was assisted by 171 external reviewers. For the first time in the history of DISC, we used double-blind peer review: the submissions were anonymous and the PC members and external reviewers did not see the names of the authors.

The program committee decided to accept 34 regular submissions for presentation at DISC 2019. After the selection of the regular papers, the authors of 19 regular papers were invited to submit brief versions of their work, and this way we received 12 additional brief submissions. From among the 25 brief submissions the program committee decided to accept 15 brief announcements for presentation.

The committee selected the following two papers as the co-recipients of the *DISC 2019 Best Paper Award*:

- Orr Fischer and Rotem Oshman: "A Distributed Algorithm for Directed Minimum-Weight Spanning Tree"
- Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic and Dragos-Adrian Seredinschi: "Scalable Byzantine Reliable Broadcast".

DISC 2019 Best Review Award was presented to D. Ellis Hershkowitz.

This year we had eight workshops held in conjunction with DISC 2019. The following workshops were organized on October 14, 2019:

- ADGA: Workshop on Advances in Distributed Graph Algorithms
  (chair: Mohsen Ghaffari)
- ApPLIED2019: Advanced tools, programming languages, and PLatforms for Implementing and Evaluating algorithms for Distributed systems
  (chairs: Chryssis Georgiou, Yanhong Annie Liu, Miguel Matos and Elad Michael Schiller)
- BTT: Workshop on Blockchain Technology and Theory
  (chairs: Ittai Abraham, Christian Cachin, Ittay Eyal, Maurice Herlihy and Maria Potop-Butucaru)

- CELLS: Computing among Cells
(chairs: Matthias Fuegger, Adrian Kosowski, Manish Kushwaha and Thomas Nowak)

The following workshops were organized on October 18, 2019:

- DiADN: Distributed Algorithms for Dynamic Networks
(chairs: Tomasz Jurdzinski and Miguel Mosteiro)
- DCC: Workshop on Distributed Cloud Computing
(chairs: Chen Avin and Gabriel Scalosub)
- FRIDA: Formal Reasoning in Distributed Algorithms
(chairs: Swen Jacobs, Igor Konnov, Stephan Merz and Josef Widder)
- HDT: Workshop on Hardware Design and Theory
(chairs: Moti Medina and Andrey Mokhov)

I would like to thank all conference participants and everyone who contributed to DISC 2019: the authors of the submitted papers, PC members and external reviewers, keynote speakers, members of the organizing committee, workshop organizers, and members of the award committees. I would also like to thank the members of the steering committee, former chairs, our colleagues in the PODC organization and many other members of the community for their valuable assistance and suggestions, EATCS for their financial support, EasyChair administrators for help with the conference management system, and the staff at Schloss Dagstuhl – Leibniz-Zentrum für Informatik for all the hard work they did with preparing this proceedings volume.

October 2019

Jukka Suomela
DISC 2019 Program Chair

Symposium Organization

DISC, the International Symposium on Distributed Computing, is an annual forum for presentation of research on all aspects of distributed computing. It is organized in cooperation with the European Association for Theoretical Computer Science (EATCS). The symposium was established in 1985 as a biannual International Workshop on Distributed Algorithms on Graphs (WDAG). The scope was soon extended to cover all aspects of distributed algorithms and WDAG came to stand for International Workshop on Distributed AlGorithms, becoming an annual symposium in 1989. To reflect the expansion of its area of interest, the name was changed to DISC (International Symposium on DIStributed Computing) in 1998, opening the symposium to all aspects of distributed computing. The aim of DISC is to reflect the exciting and rapid developments in this field.

Program Chair

Jukka Suomela — Aalto University, Finland

Program Committee

Dan Alistarh	IST Austria, Austria
Leonid Barenboim	Open University of Israel, Israel
Petra Berenbrink	University of Hamburg, Germany
Janna Burman	Université Paris-Sud (Saclay), LRI, France
Keren Censor-Hillel	Technion, Israel
Shiri Chechik	Tel-Aviv University, Israel
Colin Cooper	King's College London, UK
Jurek Czyzowicz	UQO, Canada
Carole Delporte-Gallet	IRIF, Université Paris Diderot, France
Shlomi Dolev	Ben-Gurion University of the Negev, Israel
Robert Elsaesser	University of Salzburg, Austria
Antonio Fernandez Anta	IMDEA Networks Institute, Madrid, Spain
Vijay K. Garg	University of Texas at Austin, USA
George Giakkoupis	Inria, Rennes, France
Seth Gilbert	NUS, Singapore
Wojciech Golab	University of Waterloo, Canada
Rachid Guerraoui	EPFL, Switzerland
Magnus M. Halldorsson	Reykjavik University, Iceland
Keijo Heljanko	University of Helsinki, Finland
Mark Jelasity	University of Szeged, Hungary
Tomasz Jurdzinski	University of Wroclaw, Poland
Dariusz Kowalski	Augusta University, USA, and USWPS, Poland
Christoph Lenzen	MPI for Informatics, Germany
Gopal Pandurangan	University of Houston, USA
Merav Parter	Weizmann Institute, Israel
Sriram V. Pemmaraju	University of Iowa, USA
Maria Potop-Butucaru	Sorbonne University, France
Sergio Rajsbaum	UNAM, Mexico

33rd International Symposium on Distributed Computing (DISC 2019).
Editor: Jukka Suomela

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Binoy Ravindran	Virginia Tech, USA
Andrea W. Richa	Arizona State University, USA
Peter Robinson	McMaster University, Canada
Eric Ruppert	York University, Canada
Stefan Schmid	University of Vienna, Austria
Michael Spear	Lehigh University, USA
Alexander Spiegelman	VMware Research Group, USA
Jukka Suomela (chair)	Aalto University, Finland
Jara Uitto	ETH Zurich, Switzerland
Yukiko Yamauchi	Kyushu University, Japan

Steering Committee

Cyril Gavoille	Bordeaux U., France
Fabian Kuhn	U. Freiburg, Germany
Yoram Moses (chair)	Technion, Israel
Merav Parter	Weizmann Institute, Israel
Andréa Richa (vice chair)	Arizona State U., USA
Ulrich Schmid	TU Wien, Austria
Jukka Suomela	Aalto U., Finland

Organizing Committee

András Pataricza (general chair)	BME, Hungary
Zoltán Micskei (general co-chair)	BME, Hungary
Moti Medina (workshop chair)	Ben-Gurion University of the Negev, Israel
Dennis Olivetti (proceedings chair)	Aalto University, Finland
Gábor Huszerl (local organization chair)	BME, Hungary
Attila Varga (financial chair)	Diamond Congress Ltd., Hungary
Ákos Hajdu (web chair)	BME, Hungary

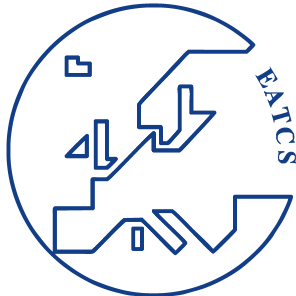
External Reviewers

Vitaly Aksenov
Yair Amir
Yackolley Amoussou Guenou
Balaji Arun
Gal Asa
James Aspnes
Hagit Attiya
John Augustine
Evangelos Bampas
Gregor Bankhamer
Joffroy Beauquier
Soheil Behnezhad
Marianna Belotti
Ran Ben Basat
Silvia Bonomi
Quentin Bramas
Armando Castañeda
Yi-Jun Chang
Themistoklis Charalambous
Bapi Chatterjee
Himanshu Chauhan
Luis F. Chiroque
Bogdan Chlebus
Vicent Cholvi
Andrea Clementi
Graham Cormode
Gianlorenzo D'Angelo
Peter Davies
Joshua Daymude
Murilo de Lima

Gianluca De Marco
Oksana Denysyuk
Michal Dory
Fabien Dufoulon
Jérôme Durand-Lose
Yuval Efron
Pavlos Eirinakis
Yuval Emek
Panagiota Fatourou
Hugues Fauconnier
Manuela Fischer
Orr Fischer
Paola Flocchini
Sebastian Forster
Pierre Fraigniaud
Florian Furbach
Matthias Függer
Mohit Garg
Leszek Gasieniec
Natalia Gavrilenko
Michał Gańczorz
Rati Gelashvili
Konstantinos Georgiou
Robert Gmyr
Gramoz Goranci
Guy Goren
Themis Gouleakis
Ofer Grossman
Joe Halpern
David Harris
Ahmed Hassan
Danny Hendler
Maurice Herlihy
D. Ellis Hershkowitz
Yael Hitron
Martin Hoefer
Changyong Hu
Damien Imbs
Tanmay Inamdar
Taisuke Izumi
Colette Johnen
Dominik Kaaser
Shahin Kamali
Pankaj Khanchandani
Ryan Killick
Valerie King
Peter Kling
Marek Klonowski
Eleftherios Kokoris-Kogias
Boris Koldehofe
Christian Konrad
Kishori Konwar
Janne H. Korhonen
Amos Korman
R. Krithika
Piotr Krysta
Fabian Kuhn
Sweta Kumari
Petr Kuznetsov
Yujin Kwon
Anissa Lamani
Jason Li
Zvi Lotker
Altti Maarala
Dahlia Malkhi
Alex Mat
Alexandre Maurer
Yannic Maus
Moti Medina
Hammurabi Mendes
Othon Michail
Jaroslaw Mirek
Slobodan Mitrovic
Neeraj Mittal
Anisur Rahaman Molla
William K. Moses Jr.
Achour Mostéfaoui
Cameron Musco
Giorgi Nadiradze
Lars Nagel
Danupon Nanongkai
Lata Narayanan
Saket Navlakha
Ofer Neiman
Calvin Newport
Nicolas Nicolaou
Ruslan Nikolaev
Thomas Nowak
Dennis Olivetti
Krzysztof Onak
Fukuhito Ooshita
Sean Ovens
Shreyas Pai
Marius Paraschiv
Ami Paz
Paolo Penna

Sathya Peri
Matthieu Perrin
Nguyen Dinh Pham
Julian Portmann
Alexandros Psomas
Mikaël Rabie
Tomasz Radzik
Thibault Rieutord
Nicolás Rivera
Will Rosenbaum
Joel Rybicki
Jared Saia
Thomas Sauerwald
Laura Schmid
Gregory Schwartzman
Michele Scquizzato
Eric Severson
Gokarna Sharma
Shantanu Sharma
Rong Shi
Masahiro Shibata
Sebastian Siebertz
Antti Tapani Siirtola
Hsin-Hao Su
Yuichi Sudo
He Sun
Nathalie Sznajder
Gadi Taubenfeld
Chris Thachuk
Tigran Tonoyan
Przemysław Uznański
Ingo van Duijn
Yadu Vasudev
Giovanni Viglietta
Koichi Wada
David Wajc
Ziyu Wang
Ben Wiederhake
Kyrill Winkler
Philipp Woelfel
Sorrachai Yingchareonthawornchai
Eylon Yogev
Maxwell Young
Igor Zablotchi
Xiong Zheng

Sponsoring Organizations



DISC is organized in cooperation with the European Association for Theoretical Computer Science (EATCS)

DISC 2019 acknowledges the use of the EasyChair system for handling submissions and managing the review process, and LIPIcs for producing and publishing the proceedings.

2019 Edsger W. Dijkstra Prize in Distributed Computing

The Edsger W. Dijkstra Prize in Distributed Computing is awarded for outstanding papers on the principles of distributed computing, whose significance and impact on the theory or practice of distributed computing have been evident for at least a decade. It is sponsored jointly by the ACM Symposium on Principles of Distributed Computing (PODC) and the EATCS Symposium on Distributed Computing (DISC). The prize is presented annually, with the presentation taking place alternately at PODC and DISC.

The committee decided to award the 2019 Edsger W. Dijkstra Prize in Distributed Computing to

Alessandro Panconesi and Aravind Srinivasan

for their paper

Randomized Distributed Edge Coloring via an Extension of the Chernoff–Hoeffding Bounds, *SIAM Journal on Computing*, volume 26, number 2, 1997, pages 350–368.

A preliminary version of this paper appeared as “Fast Randomized Algorithms for Distributed Edge Coloring”, *Proceedings of the Eleventh Annual ACM Symposium Principles of Distributed Computing (PODC)*, 1992, pages 251–262.

The paper presents a simple synchronous algorithm in which processes at the nodes of an undirected network color its edges so that the edges adjacent to each node have different colors. It is randomized, using $1.6\Delta + O(\log^{1+\delta} n)$ colors and $O(\log n)$ rounds with high probability for any constant $\delta > 0$, where n is the number of nodes and Δ is the maximum degree of the nodes. This was the first nontrivial distributed algorithm for the edge coloring problem and has influenced a great deal of follow-up work. Edge coloring has applications to many other problems in distributed computing such as routing, scheduling, contention resolution, and resource allocation.

In spite of its simplicity, the analysis of their edge coloring algorithm is highly nontrivial. Chernoff–Hoeffding bounds, which assume random variables to be independent, cannot be used. Instead, they develop upper bounds for sums of negatively correlated random variables, for example, which arise when sampling without replacement. More generally, they extend Chernoff–Hoeffding bounds to certain random variables they call λ-correlated. This has directly inspired more specialized concentration inequalities. The new techniques they introduced have also been applied to the analyses of important randomized algorithms in a variety of areas including optimization, machine learning, cryptography, streaming, quantum computing, and mechanism design.

2019 Award Committee:
Lorenzo Alvisi, *Cornell University*
Shlomi Dolev, *Ben Gurion University*
Faith Ellen (chair), *University of Toronto*
Idit Keidar, *Technion*
Fabian Kuhn, *University of Freiburg*
Jukka Suomela, *Aalto University*

33rd International Symposium on Distributed Computing (DISC 2019).
Editor: Jukka Suomela

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2019 Principles of Distributed Computing Doctoral Dissertation Award

The winner of the 2019 Principles of Distributed Computing Doctoral Dissertation Award is

Dr. Sepehr Assadi

for his dissertation

Combinatorial Optimization on Massive Datasets: Streaming, Distributed, and Massively Parallel Computation,

written under the supervision of Prof. Sanjeev Khanna at the University of Pennsylvania.

The thesis resolves a number of long-standing problems in the exciting and still relatively new area of sublinear computation. The area of sublinear computation focuses on design of algorithms that use sublinear space, time, or communication to solve global optimization problems on very large datasets. In addition to addressing a wide range of different problems, comprising graph optimization problems (matching, vertex cover, and connectivity), submodular optimization (set cover and maximum coverage), and resource-constrained optimization (combinatorial auctions and learning), these problems are studied in three different models of computation, namely, streaming algorithms, multiparty communication, and massively parallel computation (MPC). The thesis also reveals interesting relations between these different models, including generic algorithmic and analysis techniques that can be applied in all of them.

For many fundamental optimization problems, the thesis gives asymptotically matching algorithmic and intractability results, completely resolving several long-standing problems. This is accomplished by using a broad spectrum of mathematical methods in very detailed and intricate proofs. In addition to a wide variety of classic techniques, ranging from graph theory, combinatorics, probability, linear algebra and calculus, it also makes heavy use of communication complexity and information theory, for example.

Sepehr's dissertation work has been published in a remarkably large number of top-conference papers. It received multiple best paper awards and multiple special issue invitations, as well as two invitations to the Highlights of Algorithms (HALG) conference. Due to its contributions to the field of distributed computing and all the merits mentioned above, the award committee unanimously selected this thesis as the winner of the 2019 Principles of Distributed Computing Doctoral Dissertation Award.

The award is sponsored jointly by the ACM Symposium on Principles of Distributed Computing (PODC) and the EATCS Symposium on Distributed Computing (DISC). It is presented annually, with the presentation taking place alternately at PODC and DISC. The 2019 award was presented at PODC 2019 in Toronto, Canada.

2019 Award Committee:
Prasay Jayanti, *Dartmouth College*
Nancy A. Lynch, *MIT*
Boaz Patt-Shamir, *Tel Aviv University*
Ulrich Schmid, chair, *TU Wien*

33rd International Symposium on Distributed Computing (DISC 2019).
Editor: Jukka Suomela

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Consensus with Max Registers

James Aspnes
Department of Computer Science, Yale University, New Haven, CT, USA
james.aspnes@gmail.com

He Yang Er
Department of Computer Science, Yale University, New Haven, CT, USA
erheyang@gmail.com

Abstract

We consider the problem of implementing randomized wait-free consensus from max registers under the assumption of an oblivious adversary. We show that max registers solve m-valued consensus for arbitrary m in expected $O(\log^* n)$ steps per process, beating the $\Omega\left(\frac{\log m}{\log\log m}\right)$ lower bound for ordinary registers when m is large and the best previously known $O(\log\log n)$ upper bound when m is small. A simple max-register implementation based on double-collect snapshots translates this result into an $O(n \log n)$ expected step implementation of m-valued consensus from n single-writer registers, improving on the best previously-known bound of $O(n \log^2 n)$ for single-writer registers.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases consensus, max register, single-writer register, oblivious adversary, shared memory

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.1

Funding *James Aspnes*: Supported in part by NSF grants CCF-1637385 and CCF-1650596.

1 Introduction

Most work on randomized wait-free shared-memory consensus, dating back to the initial papers of Abrahamson [1] and Chor, Israeli, and Li [13], assumes either single-writer or multi-writer atomic registers as the base object provided by the system. Atomic registers have **consensus number** 1 [19], meaning they can solve consensus deterministically for no more than one process [21]. They are also assumed to be available by default when computing the consensus number of an object [19, 20]. This makes them the weakest object that is usually considered when solving shared-memory consensus. But there are other objects that also have consensus number 1 that may provide better performance than registers when implementing *randomized* consensus for many processes.

We examine the power of consensus protocols built from **max registers**, which are register-like objects for which the `writeMax` operation writes values, and the `readMax` operation returns the largest value previously written [6]. We show that against an oblivious adversary, the expected individual step complexity of binary consensus can be improved from $O(\log\log n)$ [5] using standard registers to $O(\log^* n)$ using max registers, closing the gap between the best previously known upper bounds for consensus and the closely-related problem of test-and-set [16].

Consensus protocols built from max registers also tolerate more than two possible input values better than those built from atomic registers. We show that max registers can implement an m-valued **adopt-commit object** [2, 15, 22] in $O(1)$ steps using $O(1)$ max registers. Adopt-commit objects can be derived from any shared-memory consensus protocol, and there is a tight $\Omega(\log m/\log\log m)$ lower bound on the individual step complexity of implementations of adopt-commit objects from atomic registers that carries over to consensus [8]. Using max registers, we can avoid this lower bound and obtain m-valued consensus for any m while still getting $O(\log^* n)$ individual step complexity.

© James Aspnes and He Yang Er;
licensed under Creative Commons License CC-BY
33rd International Symposium on Distributed Computing (DISC 2019).
Editor: Jukka Suomela; Article No. 1; pp. 1:1–1:9

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

By implementing a max register from single-writer registers accessed via collects (in which a process sequentially reads n registers, one for each process), we can get consensus for single-writer registers in $O(n \log n)$ expected steps per process. This improves on the previous best known $O(n \log^2 n)$ upper bound [10] for single-writer registers, although at the cost of weakening the adversary from adaptive to oblivious. It also gets close to the $\Omega(n)$ lower bound on the worst-case expected individual step complexity that follows immediately from the need for the first process to check the registers belonging to all other processes. We suspect that further improvements may be possible through a more careful analysis of how maximum values interact with collects, but this analysis is beyond the scope of the present work.

1.1 Model

We consider the standard model of an asynchronous shared-memory system, where a collection of n **processes** communicate by applying operations to shared **objects**. Typically these objects are **atomic registers**, which support a `write`(v) operation that changes the state of the register to v (and returns nothing) and a `read` operation that returns the value of the most recent previous `write` operation, or a default initial value if there is no previous `write` operation. But we will also consider **max registers**, in which a `readMax` operation instead returns the *largest* value of any preceding `writeMax` operation, or a default initial value if there is none.

Concurrency is modeled by interleaving. An **execution** $C_0\pi_1C_1\pi_2\ldots$ consists of an alternating sequence of **configurations** C_i giving the state of all processes and objects in the system, and operations π_i, each of which is carried out by some process. An operation π is **enabled** in a configuration C if there is a process in C that chooses π as its next operation. The effect of applying the operation is to transition to a new configuration C' that updates the state of the process executing π and the object to which it is applied. At each step, an **adversary** chooses which of the enabled operations occurs.

For deterministic protocols with known initial configuration C_0, an execution can be summarized by giving a **schedule** $\pi_1\pi_2\ldots$, consisting only of the operations.

For randomized protocols, an **adaptive** adversary chooses π_i with full knowledge of C_i, including internal states of the processes. An **oblivious** adversary instead fixes a sequence of process ids $p_{i_1}p_{i_2}\ldots$ at the start of the execution, and p_{i_j} carries out the j-th step in all executions. In both cases, the execution obtained is a random variable that depends on both the adversary's strategy and the outcomes of any random choices made by the processes.

The **total step complexity** of an execution is the number of operations carried out by all processes during the execution. The **individual step complexity** is the maximum number of operations carried out by any individual process during the execution.

The **space complexity** of a protocol is the total number of objects in the system, whether they are used in a particular execution or not. This does not depend on the capacity of each object: a register that holds one bit and a register that holds an unbounded number of bits both count as a single object for this purpose. The **allocate-on-use space complexity** [9] of an execution is the number of objects to which at least one operation is applied during the execution.

2 Consensus using max registers

Consensus [23] has three requirements:

Termination All processes finish.

Validity The output value of every process is the input value of some process.

Agreement All processes return the same output value.

We will be building **wait-free** consensus, meaning that the termination condition must hold for each process in all executions, regardless of the relative speeds of the other processes.

For our consensus protocol, we follow the modular construction suggested in [4]. This construction alternates **conciliator** objects with **adopt-commit** objects, and gives step complexity and space complexity proportional to the sum of the corresponding complexities of the conciliator and adopt-commit objects.

A **conciliator** object [4] is similar to a consensus protocol, but it replaces the agreement condition with a **probabilistic agreement** condition, that says that all processes return the same output with some minimum probability $\delta > 0$.

An **adopt-commit** object [3, 22] abstracts the adopt-commit protocol of Gafni [15], which is used to detect and enforce agreement obtained through other means. In addition to an output value, it also supplies a tag adopt or commit. Termination and validity hold as in consensus, but the agreement condition is now replaced by two new conditions:[1] **convergence**, which says that if all inputs are equal to some value v then all processes return $\langle \mathsf{commit}, v\rangle$; and **coherence**, which says that if any process returns $\langle \mathsf{commit}, v\rangle$ then all processes return either $\langle \mathsf{commit}, v\rangle$ or $\langle \mathsf{adopt}, v\rangle$. The significance of these conditions is that once all processes have the same input (for example, as the result of a preceding conciliator successfully producing agreement), then all will commit, and if any process commits even without all process having the same input, it will force all processes to agree on their outputs, allowing the first process to safely return knowing that any others will return the same value after the next adopt-commit.

Each execution of a conciliator has a δ chance of resulting in agreement, meaning that a protocol using this construction finishes in $1/\delta + 1$ expected phases. For constant δ, this makes the expected cost of the consensus protocol a constant multiple of the sum of the costs of the conciliator and adopt-commit objects. So our goal will be to use max registers to make both of these objects cost as little as possible.

2.1 Conciliator using max registers

To implement a conciliator from max registers, we construct a **sifter** [2] supplemented by the **persona** construction of [5]. A sifter is a mechanism for getting rid of processes quickly. The persona construction allows losers in this process to effectively take over the role of winners by copying sufficient control information (the *persona*) to duplicate the winning process's behavior.

The intuition behind our construction is that if a sequence of s distinct random ranks is written to a max register, only the left-to-right maxima will actually appear in the register. If the ranks are chosen independently of each other and of the order of the writes, there are $\sum_{i=1}^{s} \frac{1}{i} = H_s \leq 1 + \ln s$ such maxima on average, and we get a reduction from s to $O(\log s)$ values each time we apply this trick. This remains true if correlated duplicate ranks appear in the sequence, so long as the random ranks are independent of the order in which they *first* appear. This is enforced by having each process associate a sequence of independent random ranks (the *persona*) with its input value at the start of the protocol. When another process adopts the same value, it also adopts the same random ranks. This allows a fast process to pick up the value of a slow process and carry it to victory – a necessary condition for wait-freedom, since processes cannot leave until a winner is determined – without creating correlations that the adversary can exploit in the sifter mechanism.

[1] Our choice of names for these conditions follows [8].

Pseudocode for the conciliator is given in Algorithm 2.1. It uses $\ell = 4 + \log^* n$ rounds, with an independent random rank associated with each process's input value for each round. The ranks can be chosen from a variety of distributions, but for simplicity we assume that each rank is an independent uniform random integer in the range 1 through n^3.

Algorithm 2.1 Conciliator using max registers.

```
1 procedure conciliator(v)
2   Choose random ranks r_1 ... r_ℓ
3   for i ← 1 ... ℓ do
4     writeMax(M_i, ⟨r_i, ..., r_ℓ, v⟩)
5     ⟨r_i, ..., r_ℓ, v⟩ ← readMax(M_i)
6   end
7   return v
8 end
```

▶ **Lemma 1.** *Algorithm 2.1 implements a conciliator with a constant probability of agreement for sufficiently large n.*

Proof. Termination and validity are immediate from inspection of the code. Probabilistic agreement requires more work.

Define the persona of process p after 0 rounds as its initial tuple $\langle r_1, \dots, r_\ell, v\rangle$ and after i rounds as the tuple $\langle r_{i+1}, \dots, r_\ell, v\rangle$ that it stores after completing its i-th `readMax` operation in Line 5.

Let D be the event that all r_i are distinct from each other for each i. For r_i chosen uniformly in the range 1 through n^3 we will have $\Pr\left[\overline{D}\right] \le n^{-3}\binom{n}{2}\ell = O(n^{-1}\log^* n)$ by the union bound.

Let X_i be the number of distinct personae that appear across all processes after round i. We will argue by induction that $\mathrm{E}\left[X_i \mid D\right] \le f^{(i)}(n)$, where $f(x) = 1 + \ln x$ and $f^{(i)}$ is the i-fold iteration of f defined by $f^{(0)}(x) = x$ and $f^{(i+1)}(x) = f(f^{(i)}(x))$.

Initially, $\mathrm{E}\left[X_0 \mid D\right] = n = f^0(n)$. For the induction step, observe that any persona after $i > 0$ rounds is read from M_i and thus must be a left-to-right maximum of the values written to M_i. The adversary is oblivious, so it cannot observe r_i when scheduling the operations on M_i; at best, it knows only which processes share the same persona. But any particular persona $\langle r_i, \dots, r_\ell, v\rangle$ is a left-to-right maximum if and only if the first occurrence of this persona in the sequence is a left-to-right maximum. There are X_{i-1} such distinct personae, and by symmetry a persona that appears first after $j-1$ other distinct personae has a $1/j$ chance of being a left-to-right maximum. This gives $\mathrm{E}\left[X_i \mid D, X_{i-1}\right] = \sum_{j=1}^{X_{i-1}} \frac{1}{j} = H_{X_{i-1}} \le 1 + \ln X_{i-1} = f(X_{i-1})$. Because f is concave, Jensen's inequality gives $\mathrm{E}\left[X_i \mid D\right] = \mathrm{E}\left[\mathrm{E}\left[X_i \mid D, X_{i-1}\right] \mid D\right] \le \mathrm{E}\left[f(X_{i-1}) \mid D\right] \le f(\mathrm{E}\left[X_{i-1} \mid D\right]) = f(f^{(i-1)}(n)) = f^{(i)}(n)$.

A straightforward numerical calculation shows that $f(x) \le \log_2 x$ for $x \ge 10$, which implies that $f^{(i+1)}(n) \le \log_2^{(i+1)}(n)$ as long as $f^{(i)}(n) \ge 10$. It follows that $f^{(i)}(n) < 10$ when $i = \log^* n$. Applying f four more times gives $f^{(4+\log^* n)} < f^{(4)}(10) < 1.59$. So $\mathrm{E}\left[X_\ell \mid D\right] < 1.59$.

Because X_i is always at least 1, $X_\ell - 1 \ge 0$, so Markov's inequality applies to $X_\ell - 1$, giving $\Pr\left[X_\ell \ge 2 \mid D\right] = \Pr\left[X_\ell - 1 \ge 1 \mid D\right] \le \mathrm{E}\left[X_\ell - 1 \mid D\right] < 0.59$. We can remove the conditioning to get $\Pr\left[X_\ell \ge 2\right] \le 0.59 + \Pr\left[\overline{D}\right] = 0.59 + o(1)$. This completes the proof of probabilistic agreement. ◀

2.2 Adopt-commit using max registers

Pseudocode for an adopt-commit object implemented from max registers is given in Algorithm 2.2. The structure is similar to the implementation of adopt-commit from a generic **conflict detector** given in [8, Algorithm 2].

Here a pair of max registers min and max serve as the conflict detector, which detects when multiple distinct values have arrived at the adopt-commit. The mechanism for this is that min holds the smallest input value (negated to turn the max register into a min register), while max holds the largest, so that distinct values will cause these to diverge.

To obtain a commit, a process must successfully write its value to the proposal register before any process with a distinct value finishes writing to both max registers. This will cause later processes to see and adopt the same value.

Algorithm 2.2 Adopt-commit using max registers.

```
1  procedure adoptCommit(v)
2      writeMax(min, −v)
3      writeMax(max, v)
4      if proposal ≠ ⊥ then
5          v ← proposal
6      end
7      proposal ← v
8      if readMax(min) = −v and readMax(max) = v then
9          return ⟨commit, v⟩
10     else
11         return ⟨adopt, v⟩
12     end
13 end
```

▶ **Lemma 2.** *Algorithm 2.2 implements an adopt-commit object.*

Proof. Termination and validity are immediate from inspection of the code.

For convergence, suppose that all calls to adoptCommit have the same input v. Then no value other than v is ever written to proposal or max, and no value other than $-v$ is ever written to min. It follows that (a) every process has value v when it reaches Line 8, and (b) all processes read $-v$ from min and v from max in this line, causing them to return $\langle \mathsf{commit}, v \rangle$.

For coherence, suppose that some process p returns $\langle \mathsf{commit}, v \rangle$. Then at the point where p starts executing Line 8, proposal $\neq \perp$, Process p subsequently reads $-v$ from min and v from max, which implies that at this same point no process p' with a value $v' < v$ has yet written to min and no process with a value $v' > v$ has yet written to max. Since a process must do both before checking proposal in Line 4, any such process will read v from proposal and adopt this value. ◀

The proposal register is assumed to be a standard atomic register, but if a max-register-only implementation is desired, proposal can be replaced by a max register without affecting the complexity or proof of correctness.

2.3 Full result

Combining Lemmas 1 and 2 gives:

▶ **Theorem 3.** *Alternating Algorithms 2.1 and 2.2 implements randomized consensus for any number of input values against an oblivious adversary, with expected individual step complexity $O(\log^* n)$ and expected space complexity $O(\log^* n)$ in the allocate-on-use model.*

3 Consensus using collects

If max registers are not available, we can replace them with an array of n ordinary atomic registers over which we perform **collect** operations, a non-atomic operation that involves a process reading all n registers in some arbitrary order. We use a variant of the classic double-collect technique [24] to ensure that the value returned is in fact the maximum value in the array at some point during the execution. We will show that this does not impose too much additional cost and does not create opportunities for the adversary to bias the outcome.[2]

For the adopt-commit object, we skip the max register implementation entirely and simply use Gafni's original implementation of adopt-commit from two collects [15, §4.2].

Pseudocode for the conciliator is given in Algorithm 3.1. We assume that each $A[i][j]$ is an atomic register that initially holds a default value $\perp$ that is treated as smaller than all non-default values.

Algorithm 3.1 Conciliator using collect. Code for process j.

```
1  procedure conciliator(v)
2      Choose random ranks r_1 ... r_ℓ
3      for i ← 1 ... ℓ do
4          A[i][j] ← ⟨r_i, ..., r_ℓ, v⟩
5          cur ← ⊥
6          repeat
7              prev ← cur
               // Perform collect on A[i]
8              cur ← A[i][1 ... n]
9          until prev ≠ ⊥ and max(cur) = max(prev)
10         ⟨r_i, ..., r_ℓ, v⟩ ← max(cur)
11     end
12     return v
13 end
```

[2] An alternative might be to use an existing implementation of bounded max registers from atomic registers, such as the original linearizable implementation of [6] or the strongly linearizable implementation of [18]. This runs into two difficulties. First, these implementations assume multi-writer registers, and there are implementations of conciliators directly from multi-writer registers that run in $O(\log \log n)$ steps [5], much less than the $\Omega(\log n)$ steps needed for a single operation of a polynomially-bounded max register [6]. Second, neither linearizability nor strong linearizability is enough to guarantee that an implementation can be used in place of the original atomic object without affecting the behavior of the algorithm when scheduling is controlled by an oblivious adversary [17]. Though the stronger condition of **uniform linearizability** can allow such composition [14], we are not aware of any uniformly linearizable implementations of max registers.

To show that this works, we must first argue that the maximum value computed in Line 10 corresponds to the actual maximum value in $A[i][1 \dots n]$ at some point in the execution.

▶ **Lemma 4.** *Let $x = \langle r_i, \dots, r_\ell, v\rangle$ be computed in Line 10 during some execution of Algorithm 3.1. Then at some point during the execution, x is the maximum value in registers $A[i][1]$ through $A[i][n]$.*

Proof. First observe that each $A[i][j]$ can only increase over time, and thus the same holds for the maximum value in $A[i]$.

Because x is the maximum of the non-overlapping collects that provided prev and cur, we know that (a) the maximum value at the end of the first collect is at least x, and (b) the maximum value at the start of the second collect is at most x. So throughout the interval between these collects, the maximum value is exactly x. ◀

Applying the same argument as in Lemma 1, if X_i distinct personae with distinct random ranks appear across all processes after i rounds, then on average there are H_{X_i} left-to-right maxima in the sequence of values written to $A[i]$, giving $X_{i+1} \leq H_{X_i} \leq 1 + \ln X_i$ on average. Since each repetition of the inner loop of Algorithm 3.1 by some process requires a new maximum, this means that (a) on average, only $O(\log n)$ collects are performed during the first round, and (b) on average, at most $O(\log \log n)$ collects are performed during subsequent rounds. The first round dominates, giving an expected $O(\log n)$ collects per process, for an expected individual step complexity of $O(n \log n)$. We can similarly argue that after ℓ rounds there is only one surviving persona with constant probability.

As written, the algorithm uses $O(n \log^* n)$ registers. However, we can trivially have each process consolidate its $O(\log^* n)$ registers $A[1][j]$ through $A[\ell][j]$ into a single register divided into ℓ fields, giving an implementation with only n registers.

Applying the appropriate corrections to deal with the small chance of duplicate ranks, we get:

▶ **Lemma 5.** *Algorithm 3.1 implements a conciliator with constant agreement probability with expected individual step complexity $O(n \log n)$ and space complexity n.*

Since it is possible to implement adopt-commit using $O(1)$ write operations and two collects [15], and since we can consolidate all registers used by a particular process, even across rounds, into a single register, we get

▶ **Theorem 6.** *There is an implementation of randomized consensus that achieves consensus against an oblivious adversary with expected individual step complexity $O(n \log n)$ using n single-writer registers.*

The step complexity of this algorithm compares favorably with the best previously known implementation of consensus from single-writer registers, the $O(n \log^2 n)$ algorithm of [10]. It should be noted however that the price of this improvement is that we have assumed an oblivious adversary instead of an adaptive adversary.

4 Conclusion and open problems

We have shown that using max registers as a base object allows for a significant improvement in the time and space complexity of randomized consensus over previous solutions based on atomic registers. We have also shown that this approach gives improvements even for single-writer atomic registers, by implementing max registers from collects over arrays of single-writer registers.

At present there is no known non-trivial lower bound on the expected individual step complexity of randomized consensus with an oblivious adversary, even for ordinary registers, although a lower bound on the distribution of the individual step complexity is known [12]. So it may be that a more sophisticated algorithm could reduce the expected time for max-register-based consensus from $O(\log^* n)$ to as little as $O(1)$.

Similarly, there is no stronger lower bound on the individual step complexity of consensus implemented from single-writer registers than the trivial $\Omega(n)$ bound. Here we suspect that a more careful analysis of the possible set of maximum values returned by concurrent *single* collects could reduce the gap between this lower bound and the $O(n \log n)$ upper bound obtained using double collects.

It is also interesting to consider what happens in a message-passing system. It is known that the classic Attiya-Bar-Noy-Dolev (ABD) implementation of an atomic register from asynchronous message-passing with fewer than $n/2$ failures [11] extends in a straightforward way to give a linearizable implementation of a max register in $O(1)$ time using $O(n)$ messages per operation [7]. However, the same problem of linearizability interacting poorly with randomization that required using double collects in Algorithm 3.1 applies here, and it is not clear if simply using ABD in Algorithm 2.1 would prevent even an oblivious adversary from preserving more values than expected. As in the case of single collects, further analysis is needed.

References

1. Karl Abrahamson. On Achieving Consensus Using a Shared Memory. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 291–302, 1988.
2. Dan Alistarh and James Aspnes. Sub-logarithmic test-and-set against a weak adversary. In *Distributed Computing: 25th International Symposium, DISC 2011*, volume 6950 of *Lecture Notes in Computer Science*, pages 97–109. Springer-Verlag, September 2011.
3. Dan Alistarh, Seth Gilbert, Rachid Guerraoui, and Corentin Travers. Of Choices, Failures and Asynchrony: The Many Faces of Set Agreement. In Yingfei Dong, Ding-Zhu Du, and Oscar H. Ibarra, editors, *ISAAC*, volume 5878 of *Lecture Notes in Computer Science*, pages 943–953. Springer, 2009. doi:10.1007/978-3-642-10631-6_95.
4. James Aspnes. A modular approach to shared-memory consensus, with applications to the probabilistic-write model. *Distributed Computing*, 25(2):179–188, May 2012.
5. James Aspnes. Faster randomized consensus with an oblivious adversary. *Distributed Computing*, 28(1):21–29, February 2015.
6. James Aspnes, Hagit Attiya, and Keren Censor-Hillel. Polylogarithmic concurrent data structures from monotone circuits. *jacm*, 59(1):2:1–2:24, February 2012.
7. James Aspnes and Keren Censor-Hillel. Atomic Snapshots in $O(\log^3 n)$ Steps Using Randomized Helping. In Yehuda Afek, editor, *Distributed Computing: 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14–18, 2013. Proceedings*, volume 8205 of *Lecture Notes in Computer Science*, pages 254–268. Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-41527-2_18.
8. James Aspnes and Faith Ellen. Tight Bounds for Adopt-Commit Objects. *Theory of Computing Systems*, 55(3):451–474, 2014. doi:10.1007/s00224-013-9448-1.
9. James Aspnes, Bernhard Haeupler, Alexander Tong, and Philipp Woelfel. Allocate-On-Use Space Complexity of Shared-Memory Algorithms. In Ulrich Schmid and Josef Widder, editors, *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15–19, 2018*, volume 121 of *LIPIcs*, pages 8:1–8:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018. doi:10.4230/LIPIcs.DISC.2018.8.

10 James Aspnes and Orli Waarts. Randomized consensus in expected $O(n \log^2 n)$ operations per processor. *SIAM J. Comput.*, 25(5):1024–1044, October 1996.

11 Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing Memory Robustly in Message-Passing Systems. *jacm*, 42(1):124–142, 1995. doi:10.1145/200836.200869.

12 Hagit Attiya and Keren Censor-Hillel. Lower Bounds for Randomized Consensus under a Weak Adversary. *SIAM J. Comput.*, 39(8):3885–3904, 2010. doi:10.1137/090751906.

13 Benny Chor, Amos Israeli, and Ming Li. Wait-Free Consensus Using Asynchronous Hardware. *SIAM J. Comput.*, 23(4):701–712, 1994.

14 Oksana Denysyuk and Philipp Woelfel. Are Shared Objects Composable Under an Oblivious Adversary? In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC '16, pages 335–344, New York, NY, USA, 2016. ACM. doi:10.1145/2933057.2933115.

15 Eli Gafni. Round-by-Round Fault Detectors: Unifying Synchrony and Asynchrony (Extended Abstract). In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing*, pages 143–152, 1998. doi:10.1145/277697.277724.

16 George Giakkoupis and Philipp Woelfel. On the time and space complexity of randomized test-and-set. In Darek Kowalski and Alessandro Panconesi, editors, *ACM Symposium on Principles of Distributed Computing, PODC '12, Funchal, Madeira, Portugal, July 16-18, 2012*, pages 19–28. ACM, 2012. doi:10.1145/2332432.2332436.

17 Wojciech Golab, Lisa Higham, and Philipp Woelfel. Linearizable Implementations Do Not Suffice for Randomized Distributed Computation. In *Proceedings of the Forty-third Annual ACM Symposium on Theory of Computing*, STOC '11, pages 373–382, New York, NY, USA, 2011. ACM. doi:10.1145/1993636.1993687.

18 Maryam Helmi, Lisa Higham, and Philipp Woelfel. Strongly Linearizable Implementations: Possibilities and Impossibilities. In *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing*, PODC '12, pages 385–394, New York, NY, USA, 2012. ACM. doi:10.1145/2332432.2332508.

19 Maurice Herlihy. Wait-free Synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991. doi:10.1145/114005.102808.

20 Prasad Jayanti. Robust wait-free hierarchies. *J. ACM*, 44(4):592–614, 1997. doi:10.1145/263867.263888.

21 Michael C. Loui and Hosame H. Abu-Amara. Memory Requirements for Agreement Among Unreliable Asynchronous Processes. In Franco P. Preparata, editor, *Parallel and Distributed Computing*, volume 4 of *Advances in Computing Research*, pages 163–183. JAI Press, 1987.

22 Achour Mostefaoui, Sergio Rajsbaum, Michel Raynal, and Corentin Travers. The Combined Power of Conditions and Information on Failures to Solve Asynchronous Set Agreement. *sicomp*, 38(4):1574–1601, 2008.

23 M. Pease, R. Shostak, and L. Lamport. Reaching Agreement in the Presence of Faults. *jacm*, 27(2):228–234, April 1980.

24 Gary L Peterson and James E Burns. Concurrent reading while writing II: the multi-writer case. In *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*, pages 383–392. IEEE, 1987.

Putting Strong Linearizability in Context: Preserving Hyperproperties in Programs That Use Concurrent Objects

Hagit Attiya
Technion – Israel Institute of Technology, Haifa, Israel
hagit@cs.technion.ac.il

Constantin Enea
Université de Paris, IRIF, CNRS, F-75013 Paris, France
cenea@irif.fr

Abstract

It has been observed that linearizability, the prevalent consistency condition for implementing concurrent objects, does not preserve some probability distributions. A stronger condition, called *strong linearizability* has been proposed, but its study has been somewhat ad-hoc. This paper investigates strong linearizability by casting it in the context of *observational refinement* of objects. We present a strengthening of observational refinement, which generalizes strong linearizability, obtaining several important implications.

When a concrete concurrent object *refines* another, more abstract object – often sequential – the correctness of a program employing the concrete object can be verified by considering its behaviors when using the more abstract object. This means that *trace properties* of a program using the concrete object can be proved by considering the program with the abstract object. This, however, does not hold for *hyperproperties*, including many security properties and probability distributions of events.

We define *strong observational refinement*, a strengthening of refinement that preserves hyperproperties, and prove that it is *equivalent* to the existence of *forward simulations*. We show that strong observational refinement generalizes *strong linearizability*. This implies that strong linearizability is also equivalent to forward simulation, and shows that strongly linearizable implementations can be composed both horizontally (i.e., *locality*) and vertically (i.e., with *instantiation*).

For situations where strongly linearizable implementations do not exist (or are less efficient), we argue that reasoning about hyperproperties of programs can be simplified by strong observational refinement of non-atomic abstract objects.

2012 ACM Subject Classification Theory of computation → Concurrency; Theory of computation → Program specifications; General and reference → Verification

Keywords and phrases Concurrent Objects, Linearizability, Hyperproperties, Forward Simulations

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.2

Related Version Additional material can be found at `https://arxiv.org/abs/1905.12063`.

Funding *Hagit Attiya*: Partially supported by the Israel Science Foundation (1749/14 and 380/18). *Constantin Enea*: Supported in part by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 678177).

1 Introduction

Abstraction is key to the design and verification of large, complicated software. In concurrent programs, featuring intricate interactions between multiple threads, abstraction is often used to encapsulate low-level shared memory accesses within high-level abstract data types, called *concurrent objects*. Arguing about properties of such a program P is greatly simplified by

© Hagit Attiya and Constantin Enea;
licensed under Creative Commons License CC-BY

considering a concurrent object as a *refinement* of another, more abstract one: a *concrete* object O_1 is said to *observationally refine* another, *abstract* object O_2 if any behavior P can observe with O_1 is also observed by P with O_2. When O_2 is an *atomic object*, in which each operation is applied in exclusion, observational refinement is equivalent to linearizability [5, 12].[1]

Intuitively, linearizability, and more generally, observational refinement, seem to imply that anything we can prove about P with O_2 also holds when P executes with O_1. This is indeed the case when considering *trace properties*, i.e., properties that are specified as *sets of traces*, in particular, safety properties.

Unfortunately, many interesting properties cannot be specified as properties of individual traces, i.e., as trace properties. Notable examples are security properties such as *noninterference* [13], stipulating that commands executed by users with high clearance have no effect on system behavior observed by users with low clearance. Other examples are quantitative properties like bounds on the *probability distribution* of events, e.g., the mean response time over sets of executions. Indeed, while the fact that *the average response time of an operation in an execution is smaller than some bound X* is a trace property, the requirement that the *average response time over all executions is smaller than X* cannot be stated as a trace property.

Hyperproperties [9], namely, sets of sets of traces, allow to capture such expectations. By definition, every property of system behavior (for systems modeled as trace sets) can be specified as a hyperproperty. It is known that observational refinement does not preserve hyperproperties [18], in general. More recently, it has been shown that linearizability does not preserve probability distributions over traces [14], allowing an adversary scheduler additional control over the possible outcomes of a distributed randomized program. (An example appears in Section 2.)

This paper defines the notion of *strong observational refinement*, relates it to hyperproperties, and shows its equivalence to *forward simulations*. We show that strong observational refinement generalizes strong linearizability [14].[2] We also explore the possibility of using – instead of the classical sequential specifications – *concurrent* specifications, which are nevertheless simpler.

To explain our results in more detail, consider a *labeled transition system* (*LTS*) that, intuitively, represents all the executions of the object under the most general client (that may call methods in any order and from any thread). A state of the LTS corresponds to a state of the object and transitions correspond to method calls/returns, or internal steps within a method invocation. A *sequential specification* corresponds to a concurrent object where essentially, method bodies consist of a single atomic step that acts according to the sequential specification (hence, they are totally ordered in time during any execution).

An LTS O_1 *observationally refines* an LTS O_2 if and only if the *histories* (i.e., sequences of call/return actions) generated by O_1 are included in those generated by O_2 [5]. In this way, observational refinement of two LTSs reduces to a inclusion between their traces, when projected over some alphabet Γ (in this case, Γ is the set of call/return actions), called Γ-*refinement*.

A *forward simulation* maps every step of O_1 to a sequence of steps of O_2, starting from the initial state of O_1 and advancing in a forward manner; a *backward simulation* is similar, but it goes in the reverse direction, from end states back to initial states. When proving

[1] *Linearizability* [16] states that a concurrent execution of operations corresponds to some serial sequence of the same operations permitted by the specification.

[2] *Strong linearizability* requires that *the linearization of a prefix of a concurrent execution is a prefix of the linearization of the whole execution*, see Section 5.

```
a = push(0);        ||  b = push(1);        ||  assume a == b == OK;
low1 = pop();           low2 = pop();           push(2);
                                                high = highBooleanInput();
```

Figure 1 A program with three threads using a concurrent stack (we assume that `push` returns the value `OK`). Statements in the same thread are aligned vertically. The statement `assume` blocks the program when the Boolean condition is not satisfied and `highBooleanInput` returns a Boolean value labeled as high clearance. The `assume` statement enforces that `push(0)` and `push(1)` finish before `push(2)` starts.

linearizability, an important special case of forward simulation is the identification of *fixed* linearization points. A forward/backward simulation can be parameterized by an alphabet Γ, in which case the sequence of steps of O_2 associated to a step of O_1 should contain a step labeled by an action $a \in \Gamma$ if and only if the step of O_1 is also labeled by a. It is known [17] that Γ-refinement holds if and only if there is a combination of Γ-forward and Γ-backward simulations from O_1 to O_2; a forward simulation suffices when the projection of O_2 on Γ is deterministic. (See Section 3.)

The notion of strong observational refinement relies on the concept of a *deterministic scheduler* that resolves the non-determinism introduced for instance, from the execution of internal actions by parallel threads (it is similar to the notion of strong adversary introduced in the context of randomized algorithms [4]). O_1 *strongly (observationally) refines* O_2 if a program P running under a deterministic schedule with O_1 makes the same observations as when P runs with O_2 with a possibly-different deterministic schedule. (The complete definition appears in Section 4.) We prove that strong observational refinement implies the existence of a forward simulation. The converse direction is fairly straightforward, proving the equivalence of these two notions, and imply compositional proof methodologies. (These results appear in Section 5.) By relating strong linearizability to strong observational refinement, we prove that a concrete object is a strong linearization of an atomic object *if and only if* there is an appropriate forward simulation between the two. This immediately implies methods for composing strongly linearizable concurrent objects.

To address situations where there is no concrete object that strongly linearizes a particular atomic object [15, 10], or in cases where such an object is less efficient, we suggest concrete objects that strongly refine other, more abstract objects that still expose some concurrency. This follows [6] and allows to simplify reasoning about randomized programs even when using objects like the Herlihy&Wing queue [16] or snapshot objects [1], which are not strongly linearizable. For example, in the case of atomic snapshots, the abstract object that obtains several instantaneous snapshots during the *scan* operation and then *arbitrarily* returns one of them (see Section 6). Arguing about a program using this abstract object is simpler, while still exposing the power of an adversarial scheduler to manipulate the responses of a scan.

2 Motivating Example: A Stack Implementation that Leaks Information

When an object O_1 refines a specification object O_2, any safety property of a program P using O_2 (that refers only to P's state and is agnostic to the internal state of the object O_2) is preserved when O_2 is replaced by its refinement O_1. However, refinement does not preserve *hyperproperties* [9], i.e., properties of *sets* of traces.

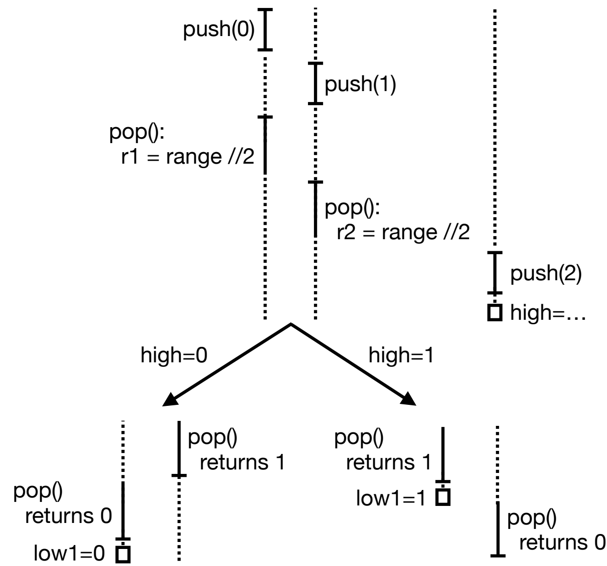


Figure 2 A scheduler for the program in Figure 1. Time flows from top to bottom. Dotted edges denote periods of time where a thread is not active.

We explain this issue by considering *noninterference* [13] in the program of Figure 1. This program invokes methods of a concurrent stack, and we wish to show that independently of the thread scheduler, none of the low clearance variables `low1` and `low2` can leak the value of the high clearance variable `high`, i.e., it is impossible to define a thread scheduler which admits only executions where `low1` = `high` or only executions where `low2` = `high`. A precise notion of scheduler will be defined below, but for now, it is enough to think of a thread scheduler as a monitor that chooses to activate threads depending on the history of the execution. This property is satisfied by the program when invoking an *atomic* concurrent stack. Indeed, assuming that `push(0)` is scheduled before the one of `push(1)` (the other case is similar), then either (i) `push(2)` is scheduled before at least one of the `pop` invocations, and then, `low1`, `low2` $\in \{1,2\}$ which shows that none of these two variables equals the value of `high` when `high` $= 0$, or (ii) `push(2)` is scheduled after the `pop` invocations, and then, the scheduler admits executions where `low1` $= b_1$, `low2` $= b_2$, and `high` $= 0$ if and only if it admits executions where `low1` $= b_1$, `low2` $= b_2$, and `high` $= 1$ (for $b_1, b_2 \in \{0,1,2\}$).

This property is however not satisfied by this program when using the concurrent stack of Afek et al. [2]. This stack stores the elements into an infinite array `items`; a shared variable `range` keeps the index of the first unused position in `items`. The push method stores the input value in the array while also incrementing `range` (the details are irrelevant for our example). The pop method first reads `range` and then traverses the array backwards starting from the predecessor of this position, until it finds a position storing a non-null element (array cells can be nullified by concurrent pop invocations). It atomically reads this element and stores null in its place. If the pop reaches the bottom of the array without finding non-null cells, then it returns that the stack is empty. Unlike the case of atomic stacks, Figure 2 shows a thread scheduler where `low1` stores the value of `high`. This scheduler imposes that `push(0)` executes before `push(1)` (so that 0 occurs before 1 in the array `items`),[3] and then

[3] A similar scheduler can be defined when `push(1)` executes before `push(0)`.

preempts pop invocations just after reading the value of `range` which equals 2 (assuming the array indexing starts at 0). Then, it schedules the third thread and, depending on the value of `high`, it schedules the rest of the `pop` invocations such that the `pop` in the first thread extracts a value which equals `high`. This ensures that `low1 == high`.

This shows that noninterference in programs invoking the atomic stack is not preserved when the latter is replaced by the concurrent stack of Afek et al. [2], although the latter is a refinement of the atomic stack. Section 4 presents a stronger notion of observational refinement that preserves hyperproperties and in particular, noninterference.

3 Modelling Concurrent Objects as Labeled Transition Systems

Labeled transition systems (*LTS*) capture shared-memory programs with an arbitrary number of threads, abstracting away the details of any particular programming system irrelevant to our development.

An LTS $A = (Q, \Sigma, s_0, \delta)$ over the possibly-infinite alphabet Σ is a possibly-infinite set Q of states with initial state $s_0 \in Q$, and a transition relation $\delta \subseteq Q \times \Sigma \times Q$. The ith symbol of a sequence $\tau \in \Sigma^*$ is denoted τ_i, and ϵ is the empty sequence. An *execution* of A is an alternating sequence of states and transition labels (also called *actions*) $\rho = s_0, a_0, s_1 \dots a_{k-1}, s_k$ for some $k > 0$ such that $(s_i, a_i, s_{i+1}) \in \delta$ for each $0 \leq i < k$. We write $s_i \xrightarrow{a_i \dots a_{j-1}}_A s_j$ as shorthand for the subsequence $s_i, a_i, ..., s_{j-1}, a_{j-1}, s_j$ of ρ. (in particular $s_i \xrightarrow{\epsilon} s_i$).

The projection $\tau|\Gamma$ of a sequence τ is the maximum subsequence of τ over alphabet Γ. This notation is extended to sets of sequences as usual. A *trace* of A is the projection $\rho|\Sigma$ of an execution ρ of A. The set of executions of an LTS A is denoted by $E(A)$, while the set of traces of A is denoted $T(A)$. An LTS is *deterministic* if for any state s and any sequence $\tau \in \Sigma^*$, there is at most one state s' such that $s \xrightarrow{\tau} s'$. More generally, for an alphabet $\Gamma \subseteq \Sigma$, an LTS is Γ-*deterministic* if for any state s and any sequence $\tau \in \Gamma^*$, there is at most one state s' such that $s \xrightarrow{\tau'} s'$ and τ is a subsequence of τ'.

An *object* is a *deterministic* LTS over alphabet $C \cup R \cup \Sigma_o$ where C is the set of call actions, R is the set of return actions, and Σ_o is an alphabet of internal actions. Formally, a call action $call(m, d, k)$ combines a method m and argument d with an operation identifier k, while a return action $ret(m, d, k)$ combines a method m and return value d with an operation identifier k. Operation identifiers are used to pair call and return actions. We assume that the traces of an object satisfy standard well-formedness properties, e.g., return actions correspond to previous call actions. Given a standard description of an object implementation as a set of methods, its LTS represents the executions of its most general client (that may call methods in any order and from any thread). The states of the LTS represent the shared state of the object together with the local state of each thread. The transitions correspond to statements in the method bodies (in which case they are labeled by internal actions in Σ_o), or call and return actions. For simplicity, we ignore the association of method invocations to threads since it is irrelevant to our development. A trace τ of an object O projected over call and return actions is called a *history* of O, and it is denoted by $hist(\tau)$. The set of histories admitted by an object O is denoted by $H(O)$. Call and return actions $call(m, _, k)$ and $ret(m, _, k)$ are called *matching* when they contain the same operation identifier. A call action is called *unmatched* in a history h when h does not contain the matching return. A history h is called *sequential* if every call $call(m, d, k)$ is immediately followed by the matching return $ret(m, _, k)$. Otherwise, it is called *concurrent*.

Linearizability [16] is a standard correctness criterion for concurrent objects expressing conformance to a given sequential specification. This criterion is based on a relation $\sqsubseteq$ between histories: $h_1 \sqsubseteq h_2$ iff there exists a well-formed history h_1' obtained from h_1 by appending

return actions that correspond to unmatched call actions in h_1 or deleting unmatched call actions, such that h_2 is a permutation of h'_1 that preserves the order between return and call actions, i.e., a given return action occurs before a given call action in h'_1 iff the same holds in h_2. We say that h_2 is a *linearization* of h_1. A history h_1 is called *linearizable* w.r.t. an object O_2 iff there exists a sequential history $h_2 \in H(O_2)$ such that $h_1 \sqsubseteq h_2$. An object O_1 is linearizable w.r.t. O_2, written $O_1 \sqsubseteq O_2$, iff each history $h_1 \in H(O_1)$ is linearizable w.r.t. O_2.

Linearizability has been shown equivalent to a criterion called *observational refinement* which states that every behavior of every program possible using a concrete object would also be possible were the abstract object used instead [5, 12] (the precise meaning of behavior is given below). Actually, this result holds only when the abstract object is *atomic*, i.e., an implementation where the methods of a sequential object are guarded by a global-lock acquisition. Formally, given a set of *sequential* histories *Seq*, an *atomic* object is an LTS $O = (Q, \Sigma, s_0, \delta)$ where the states are pairs formed of a history h and a linearization h_s of h, i.e., $Q = \{(h, h_s) : h_s \in \mathit{Seq} \text{ and } h \sqsubseteq h_s\}$, the internal actions are *linearization point* actions $\mathit{lin}(k)$ (for linearizing an operation with identifier k), i.e., $\Sigma = C \cup R \cup \{\mathit{lin}(k) : k \in \mathbb{I}\}$ where $\mathbb{I}$ denotes the set of operation identifiers, the initial state contains an empty history and linearization, $s_0 = (\epsilon, \epsilon)$, and the transition relation is defined by: $((h, h_s), a, (h', h'_s)) \in \delta$ if

$$a \in C \implies h' = h \cdot a \text{ and } h'_s = h_s$$
$$a \in R \implies h' = h \cdot a \text{ and } h'_s = h_s \text{ and } a \text{ occurs in } h'_s$$
$$a = \mathit{lin}(k) \implies h' = h \text{ and } h'_s = h_s \cdot \mathit{call}(m, d_1, k) \cdot \mathit{ret}(m, d_2, k), \text{ for some } m,\ d_1, \text{ and } d_2.$$

Call actions are only appended to the history h, return actions ensure that additionally, the linearization h'_s contains the corresponding operation, and linearization points extend the linearization with a new operation. Note that O admits every history which is linearizable w.r.t. *Seq*, i.e., $H(O) = \{h : \exists h' \in \mathit{Seq}.\ h \sqsubseteq h'\}$.

A *program* is a *deterministic* LTS over alphabet $C \cup R \cup \Sigma_p$ where Σ_p is an alphabet of *program actions*. Program actions can be interpreted for instance, as assignments to some program variables which are disjoint from the variables used by the object, or as different outcomes of a random choice. The executions of a program P with an object O are obtained as the executions of the LTS product $P \times O$ [4]. As program and object alphabets only intersect on call and return actions, our formalization supposes that programs and objects communicate only through method calls and returns, and not, e.g., through additional shared random-access memory.

Observational refinement between objects O_1 and O_2 means that any "observation" extracted from a program execution possible with O_1 (referred to as a "concrete" object), is also possible with O_2 (referred to as the "specification"). We define observations as projections of traces over program actions. The methods invoked by a program along with their inputs and return values can be recorded using additional program actions. In the context of a concrete programming language, one can use additional program variables to record the inputs before calling a method and the return values upon their return.

▶ **Definition 1.** *The object* O_1 observationally refines O_2*, written* $O_1 \leq O_2$*, iff*

$$T(P \times O_1)|\Sigma_p \ \subseteq\ T(P \times O_2)|\Sigma_p$$

for all programs P *over alphabet* $\Sigma_p \cup C \cup R$*.*

---

[4] The *product* $A_1 \times A_2$ of two LTSs is defined as usual, respecting $E(A_1 \times A_2)|(\Sigma_1 \cap \Sigma_2) = E(A_1)|\Sigma_2 \cap E(A_2)|\Sigma_1$.

The following theorem relates observational refinement to a standard notion of refinement between LTSs, defined roughly as inclusion of traces, in the context of concurrent objects.[5] For two LTSs A_1 and A_2, we say that A_1 *refines* A_2 when $T(A_1) \subseteq T(A_2)$. More generally, for an alphabet Γ, A_1 Γ-*refines* A_2 when $T(A_1)|\Gamma \subseteq T(A_2)|\Gamma$. By an abuse of notation, $A_1 \subseteq_\Gamma A_2$ denotes the fact that A_1 Γ-refines A_2 (we will omit Γ when it is understood from the context). Intuitively, the alphabet Γ represents a set of actions which are "observable" in both A_1 and A_2, the actions not in Γ are considered to be "internal" to A_1 or A_2. Observational refinement is equivalent to $(C \cup R)$-refinement which means that the histories of the concrete object are included in those of the specification (note that "plain" refinement would not hold because the internal actions may differ).

▶ **Theorem 2** ([5, 12]). *$O_1 \leq O_2$ iff $O_1 \subseteq_{C\cup R} O_2$. If O_2 is atomic, then $O_1 \leq O_2$ iff $O_1 \sqsubseteq O_2$.*

In the rest of the paper, since observational refinement and $(C \cup R)$-refinement are equivalent, we will not make the distinction between the two and refer to both as *refinement*.

4 Strong Observational Refinement

As discussed in Section 2, refinement does not preserve hyperproperties, which are properties of *sets* of traces and not individual traces as in the case of safety properties. In the following, we define a stronger notion of observational refinement that preserves such properties, using a notion of scheduler that is actually just a mechanism for resolving the non-determinism induced by internal actions, irrespectively of whether it comes from executing a set of parallel threads.

A *scheduler* for a deterministic LTS $A = (Q, \Sigma, s_0, \delta)$ over alphabet Σ is a function $S : \Sigma^* \to 2^\Sigma$ which prescribes a possible set of next actions to continue an execution based on a sequence of previous actions. A trace $\tau = a_0 \cdot \ldots \cdot a_{k-1}$ is *consistent* with a scheduler S if $a_i \in S(a_0 \cdot \ldots \cdot a_{i-1})$ for all $0 \leq i < k$ (where by an abuse of notation, $a_0 \cdot a_{-1}$ represents the empty sequence). An execution is consistent with a scheduler S if its trace is. The set of executions of an LTS A consistent with a scheduler S can be defined using an LTS which is the product between A and an LTS A_S whose states are sequences in Σ^* and the transitions link a state $\tau \in \Sigma^*$ to a state $\tau \cdot a \in \Sigma^*$ provided that $a \in S(\tau)$ (such a transition is labeled by a). Let $T(A, S)$ denote the set of traces of A consistent with S. A scheduler is *admitted* by A if for every k, if $\tau = a_0 \cdot \ldots \cdot a_{k-1}$ is a trace of A consistent with S, then $S(a_0 \cdot \ldots \cdot a_{k-1})$ is non-empty and every $a \in S(a_0 \cdot \ldots \cdot a_{k-1})$ is enabled in the state s_k with $s_0 \xrightarrow{a_0 \cdot \ldots \cdot a_{k-1}}_A s_k$.

A scheduler of an LTS $P \times O$ (the product of a program P and an object O) is called *deterministic* when it fixes in a unique way the actions of O to continue an execution, i.e., for every sequence τ, $S(\tau) \subseteq \Sigma_p$ or $|S(\tau)| = 1$ (where Σ_p is the set of program actions). When program actions represent outcomes of random choices made by the program, a deterministic scheduler can be used to model a strong adversary [4] which schedules threads depending on those outcomes. An object O_1 strongly (observationally) refines an object O_2 if any deterministic schedule admitted by a program P when using O_1 leads to exactly the same set of "observations" as a deterministic schedule admitted by P were O_2 used instead. Formally,

[5] This relationship has been shown under natural assumptions about objects and programs [5]. For instance, concerning objects, it is assumed that call actions cannot be disabled and they cannot disable other actions (they can be reordered to the left while preserving the computation), and return actions cannot enable other actions.

▶ **Definition 3.** *The object O_1* strongly (observationally) refines *O_2, written $O_1 \leq_s O_2$, iff*

for every deterministic scheduler S_1 admitted by $P \times O_1$,
there exists a deterministic scheduler S_2 admitted by $P \times O_2$,
such that $T(P \times O_1, S_1)|\Sigma_p = T(P \times O_2, S_2)|\Sigma_p$
for all programs P over alphabet $\Sigma_p \cup C \cup R$.

A *hyperproperty* of a program P over alphabet $\Sigma_p \cup C \cup R$ is a set of sets of sequences over Σ_p. For instance, the hyperproperty discussed in the context of the program in Figure 1 is the set of all sets T s.t.

$(\exists \tau \in T.\ \texttt{low1}(\tau) \neq \texttt{high}(\tau)) \wedge (\exists \tau \in T.\ \texttt{low2}(\tau) \neq \texttt{high}(\tau))$
where for any variable x, $x(t)$ is the value of x at the end of trace t. A hyperproperty φ is *satisfied* by a program P with an object O, written $P \times O \models \varphi$, if $T(P \times O, S)|\Sigma_p \in \varphi$ for every deterministic scheduler S.

▶ **Theorem 4.** *If $O_1 \leq_s O_2$, then $P \times O_2 \models \varphi$ implies $P \times O_1 \models \varphi$ for every hyperproperty φ of P.*

Proof. Assume that $O_1 \leq_s O_2$ and $P \times O_2 \models \varphi$ for some hyperproperty φ of P. Let S_1 be a deterministic scheduler admitted by $P \times O_1$. Since $O_1 \leq_s O_2$, there exists a deterministic scheduler S_2 admitted by $P \times O_2$ such that $T(P \times O_1, S_1)|\Sigma_p = T(P \times O_2, S_2)|\Sigma_p$. Since, $P \times O_2 \models \varphi$, we get that $T(P \times O_2, S_2)|\Sigma_p \in \varphi$, which implies that $T(P \times O_1, S_1)|\Sigma_p \in \varphi$. Therefore, $P \times O_1 \models \varphi$. ◀

This preservation result applies to *probabilistic* hyperproperties as well, for instance when reasoning about randomized consensus protocols [4]. Since a deterministic scheduler fixes in a unique way the object's actions to continue an execution, probability distributions can be assigned only to actions which are internal to the program P. This holds for randomized protocols, where randomization is due to coin flip operations that are internal to the protocol and do not concern the behavior of the objects it invokes. Then, the probabilities associated with program actions can be encoded in the action names, thereby encoding probabilistic (hyper)properties as properties of (sets of) traces (see [9] for more details).

5 Characterizing Strong Refinement Using Forward Simulations

In general, proving refinement between two LTSs relies on *simulation relations* which roughly, are relations between the states of the two LTSs showing that one can mimic every step of the other one. *Forward* simulations show that every outgoing transition from a given state can be mimicked by the other LTS while *backward* simulations show the same for every incoming transition to a given state. Applying induction, forward simulations show that every trace of an LTS is admitted by the other LTS starting from initial states and advancing in a forward manner, while backward simulations consider the backward direction, from end states to initial states. It has been shown that (Γ-)refinement is equivalent to the existence of a composition of forward and backward simulations, and to the existence of only a forward simulation provided that the specification[6] is (Γ-)deterministic [17]. In the following, we show that strong observational refinement is equivalent to the existence of a *forward* simulation, which implies that refinement is strictly weaker than strong observational refinement (forward simulations do not suffice to establish refinement in general).

[6] When an LTS A_1 (Γ-)refines another LTS A_2, we refer to A_2 as the specification.

▶ **Definition 5.** *Let $A_1 = (Q_1, \Sigma_1, s_0^1, \delta_1)$ and $A_2 = (Q_2, \Sigma_2, s_0^2, \delta_2)$ be two LTSs and Γ an alphabet. A relation $F \subseteq Q_1 \times Q_2$ is called a* Γ-forward simulation *from A_1 to A_2 iff $(s_0^1, s_0^2) \in F$ and:*

- *for all $s_1, s_1' \in Q_1$, $a \in \Sigma_1$, and $s_2 \in Q_2$, such that $(s_1, a, s_1') \in \delta_1$ and $(s_1, s_2) \in F$, we have that there exists $s_2' \in Q_2$ and $\tau \in \Sigma_2^*$ such that $(s_1', s_2') \in F$ and $s_2 \xrightarrow{\tau}_{A_2} s_2'$ and $\tau|\Gamma = a|\Gamma$.*

A Γ-forward simulation states that every step of A_1 is simulated by a sequence of steps of A_2 (this sequence can be empty to allow for stuttering). Since it should imply that A_1 Γ-refines A_2, every step of A_1 labeled by an observable action $a \in \Gamma$ should be simulated by a sequence of steps of A_2 where exactly one transition is labeled by a and all the other transitions are labeled by non-observable actions (this is implied by $\tau|\Gamma = a|\Gamma$). Also, every internal step of A_1 should be simulated by a sequence of internal steps of A_2.

An instantiation of forward simulations are linearizability proofs using the so-called "fixed linearization points". Linearizability of a history can be proved by showing that each invocation can be seen as happening at some point, called linearization point, occurring somewhere between the call and return actions of that invocation. Then, the linearization points are *fixed* when they are mapped to a certain fixed set of statements (usually, one statement per method). This defines a mapping between steps of a concrete implementation and steps of an atomic object, i.e., those fixed statements map to linearization point actions in the atomic object and all the other statements correspond to stuttering steps of the atomic object, thereby defining a forward simulation between the two. As a side remark, backward simulation is necessary to prove linearizability w.r.t. atomic specifications, when linearization points depend on future steps in the execution, the Herlihy&Wing queue [16] being a classic example (Schellhorn et al. [21] present such a proof).

The easier direction is showing that forward simulations imply strong refinement. A forward simulation from O_1 to O_2 can be used to simulate any scheduler S_1 of a program P using O_1 by a scheduler of the same program P when using O_2. Program actions will be replayed exactly as in S_1 while the actions of O_2 simulating actions of O_1 can be chosen according to the forward simulation.

▶ **Lemma 6.** *If there exists a $(C \cup R)$-forward simulation from O_1 to O_2, then $O_1 \leq_s O_2$.*

Proof. Let $O_1 = (Q_1, \Sigma_1, s_0^1, \delta_1)$ and $O_2 = (Q_2, \Sigma_2, s_0^2, \delta_2)$ be two objects, and F a $(C \cup R)$-forward simulation from O_1 to O_2. Let P be a program over alphabet $\Sigma_p \cup C \cup R$ and S_1 a deterministic scheduler admitted by $P \times O_1$. We define a rewriting relation $\rightsquigarrow$ between traces of $P \times O_1$ consistent with S_1 and traces of $P \times O_2$ such that intuitively, if $\tau \rightsquigarrow \tau'$ then τ' is a trace of $P \times O_2$ which simulates the trace π of $P \times O_1$ with respect to the simulation relation F. Formally, $\rightsquigarrow$ is the smallest relation satisfying the following:

- $\epsilon \rightsquigarrow \epsilon$
- if $\tau \rightsquigarrow \tau'$, $S_1(\tau) \subseteq \Sigma_p$, and $a \in S_1(\tau)$, then $\tau \cdot a \rightsquigarrow \tau' \cdot a$,
- if $\tau \rightsquigarrow \tau'$, $S_1(\tau) \cap \Sigma_p = \varnothing$, and $a = S_1(\tau)$ (in this case, $a \in \Sigma_1$ and $S_1(\tau)$ is a singleton because S_1 is deterministic), then $\tau \cdot a \rightsquigarrow \tau' \cdot F(S_1(\tau))$, where $F(S_1(\tau))$ is a sequence of actions of O_2 simulating the action $a = S_1(\tau)$ in the state reached after the trace $\tau|\Sigma_1$. Formally, if $s_0^1 \xrightarrow{\tau|\Sigma_1}_{O_1} s_1$, then a simple induction on the length of executions can show that $(s_1, s_2) \in F$ where $s_0^2 \xrightarrow{\tau'|\Sigma_2}_{O_2} s_2$. Then, since $s_1 \xrightarrow{S_1(\tau)}_{O_1} s_1'$ is a transition of O_1 and F is a forward simulation, we get that there exists s_2' such that $(s_1', s_2') \in F$ and $s_2 \xrightarrow{\sigma}_{O_2} s_2'$ and $\sigma|(C \cup R) = S_1(\tau)|(C \cup R)$. We define $F(S_1(\tau)) = \sigma$.

Then, we define a deterministic scheduler S_2 admitted by $P \times O_2$ inductively as follows:

$$S_2(\epsilon) = S_1(\epsilon) \qquad \text{if } \tau \rightsquigarrow \tau', \text{ then } S_2(\tau') = \begin{cases} S_1(\tau), \text{if } S_1(\tau) \subseteq \Sigma_p \\ F(S_1(\tau)), \text{otherwise} \end{cases}$$

Note that S_2 is a slight deviation from the definition of a scheduler because $F(S_1(\tau))$ is not necessarily a single action, but a sequence of actions. However, the definition of S_2 can be adapted easily such that this sequence of steps is performed one by one. For any other sequence τ' which is not considered in the definition above, $S_2(\tau')$ is defined arbitrarily.

Since F is a $(C \cup R)$-forward simulation, $T(P \times O_1, S_1)|\Sigma_p \subseteq T(P \times O_2, S_2)|\Sigma_p$ is obvious. The reverse, i.e., $T(P \times O_2, S_2)|\Sigma_p \subseteq T(P \times O_1, S_1)|\Sigma_p$, follows from the fact that S_2 is defined inductively following the definition of S_1. ◀

We now prove our key technical result: strong observational refinement (from O_1 to O_2) implies the existence of a $(C \cup R)$-forward simulation (from O_1 to O_2). Since the latter implies refinement, a corollary of this result is that strong observational refinement implies observational refinement. Thus, we define a program P which corresponds to the most general client (of O_1) and which uses particular program actions to guess the possible continuations of a given execution with call and return actions. Then, we define a scheduler S_1 which ensures that the executions of P with O_1 are consistent with the guesses made by the program. By strong observational refinement, there exists a scheduler S_2 such that P produces the same sequences of "guess" actions and call/return actions when using O_2 and constrained by S_2 as when using O_1 and constrained by S_1 (since strong observational refinement considers traces projected over program actions, the preservation of call/return actions is not guaranteed explicitly, but it can be enforced using additional program actions used to record them). If Γ is the union of the set of "guess" actions and the set of call/return actions, then the program P used in conjunction with O_2 and constrained by the scheduler S_2 is Γ-deterministic. Therefore, there exists a forward simulation between the two variations of P. Because the program states are disjoint from the object states, this forward simulation between programs leads to a forward simulation between objects.

▶ **Lemma 7.** *If $O_1 \leq_s O_2$, then there exists a $(C \cup R)$-forward simulation from O_1 to O_2.*

Proof. Let $O_1 = (Q_1, \Sigma_1, s_0^1, \delta_1)$ and $O_2 = (Q_2, \Sigma_2, s_0^2, \delta_2)$ be two objects. Also, let $\Sigma_p = \{record(a), guess(H) : a \in C \cup R, H \subseteq (C \cup R)^*\}$ be a set of program actions for recording a call/return action a ($record(a)$) or guessing a set H of possible continuations with sequences of call/return actions ($guess(H)$). We define a program P with a single state and self-loop transitions labeled by all symbols in $\Sigma_p \cup C \cup R$, i.e., $P = (\{s_0\}, \Sigma_p \cup C \cup R, s_0, \delta)$ where $(s_0, \alpha, s_0) \in \delta$ for all $\alpha \in \Sigma_p \cup C \cup R$.

We define a deterministic scheduler S_1 which ensures that the guesses made by P when using O_1 are correct, and that the call/return actions are tracked correctly using *record* actions. To ensure the correctness of guesses, we define a mapping $after_1 : Q_1 \to 2^{(C \cup R)^*}$ which associates every state s with the set of call/return sequences admitted from s, i.e., $after_1(s) = \{\sigma : \sigma \in (C \cup R)^*, \exists \tau, s'.\ s \xrightarrow{\tau}_{O_1} s' \wedge \tau|(C \cup R) = \sigma\}$.

Let S_1 be a deterministic scheduler such that for every $a_0, \ldots, a_{k-1} \in \Sigma_1$ and $k \geq 0$,

$$S_1(a_0 \cdot \ldots \cdot a_{k-1}) = \{record(a_{k-1})\} \text{ if } a_{k-1} \in C \cup R \text{ and } k \geq 1$$

$$S_1(a_0 \cdot \ldots \cdot a_{k-1}[\cdot record(a_{k-1})]) = \{guess(H) : \exists a.\ s_0^1 \xrightarrow{a_0 \cdot \ldots \cdot a_{k-1}|\Sigma_1}_{O_1} s \xrightarrow{a}_{O_1} s' \text{ and } H = after_1(s')\}$$
$$\text{if } a_{k-1} \notin \Sigma_p$$

$$S_1(a_0 \cdot \ldots \cdot a_{k-1} \cdot guess(H)) = \{a\}, \text{ for some } a \in \Sigma_1 \text{ s.t. } s_0^1 \xrightarrow{a_0 \cdot \ldots \cdot a_{k-1} \cdot a|\Sigma_1}_{O_1} s \text{ and } H = after_1(s)$$

Informally, the first rule enforces that every call/return action a is followed by a program action $record(a)$. The second rule ensures that S_1 is permissive enough, i.e., it allows all the successors of the current object state that have different $after_1$ images. More precisely, for every sequence σ ending in an internal object action $a_{k-1} \in \Sigma_1 \setminus (C \cup R)$ or the sequence $a_{k-1} \cdot record(a_{k-1})$ when $a_{k-1} \in C \cup R$ (we use $\sigma[\cdot a]$ to denote a sequence where the character a is optional), S_1 schedules every $guess(H)$ action where H is the $after_1$ image of a successor of the current object state. The third rule ensures that every $guess(H)$ is followed by an action leading to an object state s with $H = after_1(s)$. The last two cases ensure that every action a of O_1 is preceded by a $guess(H)$ program action where H is the set of call/return sequences admitted from the post-state of a.

Although S_1 does not admit all the executions of O_1 (because of the arbitrary choice of a in the third case above), we show that the set of executions it admits simulate all the executions of O_1: let $O_1[S_1]$ be an LTS representing the set of executions of O_1 consistent with S_1 (obtained from the set of executions of P consistent with S_1 by projecting out the program state and actions). We show that the relation F_1 between states of O_1 and $O_1[S_1]$, respectively, defined by $(s, s') \in F_1$ iff $after_1(s) = after_1(s')$, is a $(C \cup R)$-forward simulation from O_1 to $O_1[S_1]$. The fact that it relates the initial object states s_0^1 and s_0^1 is trivial. Now, let $s, s_1 \in Q_1$ and $a \in \Sigma_1$ such that $(s, a, s_1) \in \delta_1$ and $(s, s') \in F_1$. Using a simple induction on the length of executions, it can be shown that there exists a state s_1' with $after_1(s_1) = after_1(s_1')$ such that (s', b, s_1') for some action b. If $a \in C \cup R$, then $b = a$ because otherwise, the continuations with call/return actions admitted from s_1 will be different from those admitted from s_1' (for instance, if a is a call action and b is an internal action, then the matching return action will be eventually enabled in executions starting from s_1 but not from s_1', at least not before a occurs). For the same reason, if a is an internal action, then b is also an internal action. This concludes the proof that F_1 is a forward simulation.

Since $O_1 \leq_s O_2$, there exists a scheduler S_2 such that $T(P \times O_1, S_1)|\Sigma_p = T(P \times O_2, S_2)|\Sigma_p$. Let $P[O_2, S_2]$ denote the LTS representation of the set of executions of P with O_2 and consistent with S_2 (explained in Section 4). It can be easily seen that $P[O_2, S_2]$ is Σ_p-deterministic (the interleaving of a sequence of Σ_p actions with internal actions of O_2 is uniquely determined by S_2 because it is a deterministic scheduler). Let $P[O_1, S_1]$ be the LTS representation of the set of executions of P with O_1 and consistent with S_1. Since $T(P[O_1, S_1])|\Sigma_p \subseteq T(P[O_2, S_2])|\Sigma_p$,[7] we get that there exists a Σ_p-forward simulation F_{S_1,S_2} from $P[O_1, S_1]$ to $P[O_2, S_2]$. Such a forward simulation defines a relation between states of O_1 and O_2, respectively, by removing the program state, i.e., s_1 and s_2 are related whenever $((s_0, s_1), (s_0, s_2)) \in F_{S_1,S_2}$. For simplicity, this relation is denoted by F_{S_1,S_2} as well. Because of the $record(a)$ actions in Σ_p, we get that F_{S_1,S_2} is a $(C \cup R)$-forward simulation from $O_1[S_1]$ to O_2. It is easy to check that $F_1 \circ F_{S_1,S_2}$ (where $\circ$ is the usual composition of relations) is a $(C \cup R)$-forward simulation from O_1 to O_2. ◀

The two lemmas above imply that:

▶ **Theorem 8.** *$O_1 \leq_s O_2$ iff there exists a $(C \cup R)$-forward simulation from O_1 to O_2.*

The fact that forward simulations are *necessary* for strong refinement makes it possible to derive in a simple way compositional methods for proving strong refinement. In the following we consider the case of *composed* objects defined as a product of a fixed set of objects, and *parametrized* objects defined from a set of "base" objects which are considered as parameters.

[7] Note that $T(P \times O_i, S_i)$ and $T(P[O_i, S_i])$ with $i \in \{1, 2\}$ denote exactly the same set of traces.

We show that strong refinement is a *local* property, i.e., it holds for composed objects if and only if it holds for individual objects in this composition. As usual, we consider compositions of objects with disjoint states and sets of actions. Indeed, any forward simulation between composed objects can be "projected" to a set of forward simulations that hold between individual objects, and vice versa. We state this result for compositions of two objects, the extension to an arbitrary number of objects is obvious.

▶ **Theorem 9.** *Let O_1 and O_2, resp., O'_1 and O'_2, be two objects over an alphabet Σ, resp., Σ', such that $\Sigma \cap \Sigma' = \emptyset$. Then, $O_1 \times O'_1 \leq_s O_2 \times O'_2$ iff $O_1 \leq_s O_2$ and $O'_1 \leq_s O'_2$.*

Next, we consider the case of parametrized objects whose implementation is parametrized by a set of base objects, e.g., snapshot objects defined from a set of atomic registers. We show that if the parametrized object is a strong refinement of an abstract specification *Spec* assuming that the base objects behave according to their own abstract specifications $Spec_i$, then instantiating any base object with an implementation that is a strong refinement of $Spec_i$ leads to an object which remains a strong refinement of *Spec*. Assuming for simplicity only one base object, a parametrized object O can be formally defined as a product $O = Spec_1 \times C$ where $Spec_1$ is the base object's specification and C is the context in which this object is used to derive the implementation of O [8]. To distinguish parametrization from composition, we use $O(Spec_1)$ to denote an object parametrized by a base object $Spec_1$. The next result is an immediate consequence of the fact that the forward simulation admitted by the base object can be composed [9] with the one admitted by the parametrized object (assuming base object's specification) to derive a forward simulation for the instantiation.

▶ **Theorem 10.** *If $O(Spec_1) \leq_s Spec$ and $B_1 \leq_s Spec_1$, then $O(B_1) \leq_s Spec$.*

Finally, it can be shown that the existence of forward simulations is equivalent to *strong linearizability* [14] when concrete objects are related to *atomic* abstract objects. Thus, let O_2 be an atomic object defined by a set of sequential histories *Seq*, i.e., $H(O_2) = \{h : \exists h' \in Seq.\ h \sqsubseteq h'\}$ (according to the definition in Section 3). We say that an object O_1 is *strongly linearizable* w.r.t. O_2, written $O_1 \sqsubseteq_s O_2$, when there exists a function $f : T(O_1) \to Seq$ such that (1) for any trace $\tau \in T(O_1)$, $hist(\tau) \sqsubseteq f(\tau)$, and (2) f is prefix-preserving, i.e., for any two traces $\tau_1, \tau_2 \in T(O_1)$ such that τ_1 is a prefix of τ_2, $f(\tau_1)$ is a prefix of $f(\tau_2)$. It can be shown that the function f induces a forward simulation and vice-versa.

The easier direction is showing that existence of a forward simulation F implies strong linearizability. Essentially, the sequential history associated to a given trace τ (by the function f) is extracted from the atomic object state related by F with the end state of τ.

▶ **Lemma 11.** *Let O_1 be an object and O_2 an atomic object. If there exists a $(C \cup R)$-forward simulation from O_1 to O_2, then O_1 is strongly linearizable w.r.t. O_2.*

Proof. Let F be a $(C \cup R)$-forward simulation from O_1 to O_2. Also, let $state(\tau)$ denote the state of O_1 reached after a trace τ (since O_1 is deterministic, this state is unique). We define a function $f : T(O_1) \to Seq$ by $f(\tau) = h_s$ where h_s satisfies $(state(\tau), (h, h_s)) \in F$. The fact that $hist(\tau) \sqsubseteq f(\tau)$ for every trace τ follows from the definition since (h, h_s) is a valid state of O_2 and $h = hist(\tau)$ (because F preserves call and return actions). The fact that f is prefix-preserving follows from the fact that F is a forward simulation. ◀

[8] For a parametrized object $O = Spec_1 \times C$, the alphabets of $Spec_1$ and C share the call/return actions of $Spec_1$ (the base object) and the alphabet of C contains the call/return actions of O. This is different from the the composition of two objects $O_1 \times O'_1$ where the alphabets of O_1 and O'_1 are disjoint.

[9] Here, we refer to classical composition of relations.

For the reverse direction, strong linearizability implies the existence of a forward simulation where a "concrete" object state s_1 is related to an atomic object state which contains the sequential history associated by the function f witnessing strong linearizability to a trace leading to s_1.

▶ **Lemma 12.** *Let O_1 be an object and O_2 an atomic object. If O_1 is strongly linearizable w.r.t. O_2, then there exists a $(C \cup R)$-forward simulation from O_1 to O_2.*

Proof. Let F be a relation between states of O_1 and O_2 defined by $(s_1, s_2) \in F$ iff there exists a trace τ such that $s_1 = \mathit{state}(\tau)$ and $s_2 = (\mathit{hist}(\tau), f(\tau))$ (by the definition of f in strong linearizability, the latter is a valid state of O_2).

We show that F is a $(C \cup R)$-forward simulation from O_1 to O_2. The fact that it relates the initial object state s_0^1 and the initial state (ϵ, ϵ) of O_2 is trivial. Now, let $s_1, s_1' \in Q_1$, $a \in \Sigma_1$, and $s_2 \in Q_2$, such that $(s_1, a, s_1') \in \delta_1$ and $(s_1, s_2) \in F$. We have to show that there exists $s_2' \in Q_2$ and $\sigma \in \Sigma_2$ such that $(s_1', s_2') \in F$, $s_2 \xrightarrow{\sigma}_{O_2} s_2'$, and $\sigma|(C \cup R) = a|(C \cup R)$. (Let τ be a trace such that $s_1 = \mathit{state}(\tau)$ and $s_2 = (\mathit{hist}(\tau), f(\tau))$. Then, $s_1' = \mathit{state}(\tau \cdot a)$ and $f(\tau)$ is a prefix of $f(\tau \cdot a)$. Several cases are to be discussed:

- if $a \in C$, then $\mathit{hist}(\tau \cdot a) = \mathit{hist}(\tau) \cdot a$ and $\mathit{hist}(\tau) \cdot a \sqsubseteq f(\tau)$ provided that $\mathit{hist}(\tau) \sqsubseteq f(\tau)$. Therefore, $s_2 \xrightarrow{a}_{O_2} (\mathit{hist}(\tau) \cdot a, f(\tau))$ and $(s_1', (\mathit{hist}(\tau) \cdot a, f(\tau))) \in F$.
- if $a \in R$ and the operation identifier k in a occurs in $f(\tau)$, then $s_2 \xrightarrow{a}_{O_2} (\mathit{hist}(\tau) \cdot a, f(\tau))$ and $(s_1', (\mathit{hist}(\tau) \cdot a, f(\tau))) \in F$ like above. If k does not occur in $f(\tau)$, then $f(\tau \cdot a) = f(\tau) \cdot c \cdot a$ where c is the call action corresponding to a (otherwise, $f(\tau \cdot a)$ would not be a linearization of $\mathit{hist}(\tau \cdot a)$). By the definition of O_2, we have that $s_2 \xrightarrow{\mathit{lin}(k) \cdot a}_{O_2} (\mathit{hist}(\tau) \cdot a, f(\tau \cdot a))$ which concludes the proof of this case.
- if $a \notin C \cup R$, then $f(\tau \cdot a)$ is obtained from $f(\tau)$ by appending some sequence of operations with identifiers k_1, …, k_n (this follows from the fact that f is prefix-preserving). Then, $s_2 \xrightarrow{\mathit{lin}(k_1) \cdot \ldots \cdot \mathit{lin}(k_n)}_{O_2} (\mathit{hist}(\tau), f(\tau \cdot a))$ and $(s_1', (\mathit{hist}(\tau), f(\tau \cdot a))) \in F$ (because in this case, $\mathit{hist}(\tau \cdot a) = \mathit{hist}(\tau)$). ◀

Lemma 11 and Lemma 12 imply the following.

▶ **Theorem 13.** *If O_2 is atomic, then $O_1 \sqsubseteq_s O_2$ iff there exists a $(C \cup R)$-forward simulation from O_1 to O_2.*

6 Strong Observational Refinements of Non-Atomic Specifications

We demonstrate that many concurrent objects defined in the literature are strong observational refinements of much simpler abstract objects, even though not necessarily atomic (w.r.t. the definition of atomic object in Section 3). We focus on objects which are not strongly linearizable, since by Theorem 13, the latter are strong refinements of atomic objects.

Figure 3 lists an implementation of a snapshot object with two methods `update(i,data)` for writing the value `data` to a location `i` of a shared array `mem`, and `scan()` for returning a snapshot of the array `mem`.[10] While the implementation of `update` is obvious, a `scan` operation performs several "collect" phases, where it reads successively all the cells of `mem`, until two consecutive phases return the same array.

[10] This is a simplified version of the snapshot object defined by Afek et al. [1].

```
1 procedure update(i,data)
2   mem[i] = data;

4 procedure scan()
5   for i = 1 to n do r1[i] = mem[i];
6   repeat
7     r2 = r1;
8     for i = 1 to n do r1[i] = mem[i];
9   until r1 == r2
10  return r1;
```

```
1 procedure update(i,data)
2   mem[i] = data;

4 procedure scan()
5   while ( nondet )
6     r = atomic_snapshot();
7     snaps = snaps · r;
8   return r1 ∈ snaps;
```

Figure 3 A snapshot object (on the left), and a concurrent specification (on the right). The shared state of both is an array `mem` of size `n`. The local variables `r1`, `r2`, and `r` are arrays of size `n` (initialized to the same value as `mem`). The local variable `snaps` is a sequence of arrays of size `n` (· denotes the concatenation operator), initially containing a single array which equals the initial value of `mem`. The use of `nondet` means that the loop is executed for an arbitrary number of times. The procedure `atomic_snapshot` returns a snapshot of `mem` in a single step executed in isolation.

This object does *not* admit a forward simulation towards the standard atomic specification where `scan` takes a *single* instantaneous snapshot of the entire array which is subsequently returned (it is not a strong refinement of such a specification). Intuitively, this holds because the linearization point of `scan` depends on future steps in the execution, e.g., a read in the second `for` loop is a linearization point only if it is not followed by updates on array cells before and after the current loop index. This is exactly the scenario in which backward simulations are necessary, intuitively, reading an execution backwards it is possible to identify precisely the linearization points of `scan` invocations. The impossibility of defining such a forward simulation is also a consequence of this object not being strongly linearizable [14].

However, this object is a strong refinement of the simpler "concurrent" specification given on the right of Figure 3 (see more explanation in the full version). The implementation of `update` remains the same, while a `scan` operation performs a sequence of *instantaneous* snapshots of the entire array `mem` and returns *any* snapshot in this sequence. Compared to the implementation on the left, it is simpler because it does not allow that reading the array `mem` is interleaved with other operations. However, it is not atomic since an execution of `scan` contains more than one step. In comparison with the atomic specification, the sequence of snapshots in `scan` allows that an adversary (scheduler) decides on the return value "lazily" after observing other invocations, e.g., updates, exactly as in the concrete implementation. Therefore, the abstract specification in Figure 3 can be used while reasoning about hyperproperties of clients, which is not the case for the atomic specification.

Beyond snapshot objects, Bouajjani et al. [6] show that a similar simplification holds even for concurrent queues and stacks which are not strongly linearizable, e.g., Herlihy&Wing queue [16] and Time-Stamped Stack [11]. These objects admit forward simulations towards "concurrent" specifications where roughly, the elements are stored in a partially-ordered set instead of a sequence (which is consistent with the real-time order between the enqueues/-pushes that added those elements). The enqueues/pushes have no internal steps, while the dequeues/pops have a single internal step which roughly, corresponds to a linearization point that extracts a minimal (for queues) or maximal (for stacks) element from the partially-ordered set. The stack of Afek at al. [2] can also be proved to be a strong refinement of such a specification. These forward simulations imply that these objects are strong refinements of their specifications.

7 Related Work and Discussion

An important contribution of our paper is to put the work on strong linearizability [10, 14, 15] in the context of standard results concerning hyperproperties [8, 9] and property-preserving refinements [3, 17, 18]. McLean [18] showed that refinements do not preserve security properties, which were later found to be instances of the more generic notion of hyperproperty [9]. By exploiting the equivalence between linearizability and refinement [5, 12], our paper clarifies that a stronger notion of linearizability is needed because standard linearizability does not preserve hyperproperties.

Our notion of strong observational refinement is a variation of the hyperproperty-preserving refinement introduced in [9], which takes into account the specificities of concurrent object clients. The relationship between forward simulations and preservation of hyperproperties has been investigated in [3]. They show that the existence of forward simulations is *sufficient* for preserving some specific class of hyperproperties (information-flow security properties like non-interference), corresponding to the straightforward direction of Theorem 8 (Lemma 6); they also show that their condition is *not necessary* in their context. In contrast, our work shows that the existence of forward simulations is *both necessary and sufficient* for preserving any hyperproperty in the context of concurrent object clients.

An important consequence of our results is that strong linearizability is equivalent to the existence of a forward simulation towards an atomic specification. This equivalence has been established independently by Rady [20], albeit using a different formalism that leads to a relatively more complex proof. The equivalence to the well-studied notion of forward simulation immediately implies methods for composing concurrent objects, in particular, *locality* and *instantiation*. This stands in contrast to the effort needed to prove similar results in [14] and [19].

While [14] relates strong linearizability to preserving a rather unspecified class of properties of randomized programs when replacing objects by their atomic specifications, the equivalence we prove implies that strong linearizability is necessary and sufficient for preserving hyperproperties in this context. Note that forward simulations are more general than strong linearizability. Section 6 presents several objects which are *not* strongly linearizable, but which admit forward simulations towards non-atomic abstract specifications. Our results imply that it is sound to use such specifications when reasoning about hyperproperties of client programs. Moreover, as opposed to strong linearizability, forward simulations are applicable to *interval-linearizable* objects [7], which do not have any atomic specification, but are essentially LTSs as in our formalization.

Finally, Bouajjani et al. [6] show that intricate implementations of concurrent stacks and queues like Herlihy&Wing queue [16] and Time-Stamped Stack [11] admit forward simulations towards non-atomic abstract specifications, but they do not discuss the connection between existence of forward simulations and preservation of hyperproperties, which is the main contribution of our paper.

Our definition of strong observational refinement and its relation to forward simulations deepens our understanding of the role of strong linearizability in preserving hyperproperties. We plan to explore strong observational refinement of almost-atomic objects and develop additional proof methodologies. Also, our notion of strong refinement uses deterministic schedulers that model strong adversaries w.r.t. Aspnes' classification [4], and it is interesting to explore variations of this notion that take into account other adversary models.

References

1 Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic Snapshots of Shared Memory. *J. ACM*, 40(4):873–890, 1993. doi:10.1145/153724.153741.

2 Yehuda Afek, Eli Gafni, and Adam Morrison. Common2 extended to stacks and unbounded concurrency. *Distributed Computing*, 20(4):239–252, 2007. doi:10.1007/s00446-007-0023-3.

3 Rajeev Alur, Pavol Cerný, and Steve Zdancewic. Preserving Secrecy Under Refinement. In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *Automata, Languages and Programming, 33rd International Colloquium, ICALP 2006, Venice, Italy, July 10-14, 2006, Proceedings, Part II*, volume 4052 of *Lecture Notes in Computer Science*, pages 107–118. Springer, 2006. doi:10.1007/11787006_10.

4 James Aspnes. Randomized protocols for asynchronous consensus. *Distributed Computing*, 16(2-3):165–175, 2003. doi:10.1007/s00446-002-0081-5.

5 Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. Tractable Refinement Checking for Concurrent Objects. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 651–662. ACM, 2015. doi:10.1145/2676726.2677002.

6 Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Suha Orhun Mutluergil. Proving Linearizability Using Forward Simulations. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 542–563. Springer, 2017. doi:10.1007/978-3-319-63390-9_28.

7 Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. Unifying Concurrent Objects and Distributed Tasks: Interval-Linearizability. *J. ACM*, 65(6):45:1–45:42, 2018. doi:10.1145/3266457.

8 Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. Temporal Logics for Hyperproperties. In Martín Abadi and Steve Kremer, editors, *Principles of Security and Trust - Third International Conference, POST 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8414 of *Lecture Notes in Computer Science*, pages 265–284. Springer, 2014. doi:10.1007/978-3-642-54792-8_15.

9 Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010. doi:10.3233/JCS-2009-0393.

10 Oksana Denysyuk and Philipp Woelfel. Wait-Freedom is Harder Than Lock-Freedom Under Strong Linearizability. In Yoram Moses, editor, *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, volume 9363 of *Lecture Notes in Computer Science*, pages 60–74. Springer, 2015. doi:10.1007/978-3-662-48653-5_5.

11 Mike Dodds, Andreas Haas, and Christoph M. Kirsch. A Scalable, Correct Time-Stamped Stack. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 233–246. ACM, 2015. doi:10.1145/2676726.2676963.

12 Ivana Filipovic, Peter W. O'Hearn, Noam Rinetzky, and Hongseok Yang. Abstraction for concurrent objects. *Theor. Comput. Sci.*, 411(51-52):4379–4398, 2010. doi:10.1016/j.tcs.2010.09.021.

13 Joseph A. Goguen and José Meseguer. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*, pages 11–20. IEEE Computer Society, 1982. doi:10.1109/SP.1982.10014.

14 Wojciech M. Golab, Lisa Higham, and Philipp Woelfel. Linearizable implementations do not suffice for randomized distributed computation. In Lance Fortnow and Salil P. Vadhan, editors, *Proceedings of the 43rd ACM Symposium on Theory of Computing, STOC 2011, San Jose, CA, USA, 6-8 June 2011*, pages 373–382. ACM, 2011. doi:10.1145/1993636.1993687.

15 Maryam Helmi, Lisa Higham, and Philipp Woelfel. Strongly linearizable implementations: possibilities and impossibilities. In Darek Kowalski and Alessandro Panconesi, editors, *ACM Symposium on Principles of Distributed Computing, PODC '12, Funchal, Madeira, Portugal, July 16-18, 2012*, pages 385–394. ACM, 2012. doi:10.1145/2332432.2332508.

16 Maurice Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. doi:10.1145/78969.78972.

17 Nancy A. Lynch and Frits W. Vaandrager. Forward and Backward Simulations: I. Untimed Systems. *Inf. Comput.*, 121(2):214–233, 1995. doi:10.1006/inco.1995.1134.

18 John McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *1994 IEEE Computer Society Symposium on Research in Security and Privacy, Oakland, CA, USA, May 16-18, 1994*, pages 79–93. IEEE Computer Society, 1994. doi:10.1109/RISP.1994.296590.

19 Sean Ovens and Philipp Woelfel. Strongly Linearizable Implementations of Snapshots and Other Types. In *38th ACM Symposium on Principles of Distributed Computing (PODC 2019)*, 2019.

20 Amgad Sadek Rady. Characterizing Implementations that Preserve Properties of Concurrent Randomized Algorithms. Master's thesis, York University, Toronto, Canada, 2017.

21 Gerhard Schellhorn, Heike Wehrheim, and John Derrick. How to Prove Algorithms Linearisable. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 243–259. Springer, 2012. doi:10.1007/978-3-642-31424-7_21.

Long-Lived Counters with Polylogarithmic Amortized Step Complexity

Mirza Ahad Baig
LaBRI, Bordeaux INP, France
CNRS, ReLaX, UMI2000, Siruseri, India
Chennai Mathematical Institute, Siruseri, India
mirzabaig.cmi@gmail.com

Danny Hendler
Ben-Gurion University of the Negev, Beer-Sheva, Israel
hendlerd@cs.bgu.ac.il

Alessia Milani
LaBRI, Bordeaux INP, France
milani@labri.fr

Corentin Travers
LaBRI, Bordeaux INP, France
travers@labri.fr

Abstract

A shared-memory counter is a well-studied and widely-used concurrent object. It supports two operations: An `Inc` operation that increases its value by 1 and a `Read` operation that returns its current value. Jayanti, Tan and Toueg [16] proved a linear lower bound on the *worst-case* step complexity of obstruction-free implementations, from read and write operations, of a large class of shared objects that includes counters. The lower bound leaves open the question of finding counter implementations with sub-linear *amortized* step complexity.

In this paper, we address this gap. We present the first wait-free n-process counter, implemented using only read and write operations, whose amortized operation step complexity is $O(\log^2 n)$ in all executions. This is the first non-blocking read/write counter algorithm that provides sub-linear amortized step complexity in *executions of arbitrary length*. Since a logarithmic lower bound on the amortized step complexity of obstruction-free counter implementations exists, our upper bound is optimal up to a logarithmic factor.

2012 ACM Subject Classification Theory of computation → Shared memory algorithms; Theory of computation → Concurrent algorithms

Keywords and phrases Shared Memory, Wait-freedom, Counter, Amortized Complexity, Concurrent Objects

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.3

Funding Mirza Ahad Baig, Alessia Milani and Corentin Travers are supported by ANR projects Descartes and FREDDA. Mirza Ahad Baig is additionally supported by UMI Relax. Danny Hendler is supported by the Israel Science Foundation (grant 380/18).

Acknowledgements We thank the anonymous reviewers for their many helpful comments.

1 Introduction

A shared-memory *counter* [18] is a well-studied and widely-used concurrent object [2, 5, 7, 10, 17]. A counter supports two operations: An `Inc` operation that increases its value by 1 and a `Read` operation that returns its current value.

A wait-free counter can be constructed easily by using an *atomic snapshot* [1, 3, 7] object, allowing each process to update its own component (by invoking an `Update` operation) and to obtain an atomic view of all components (by invoking a `Scan` operation). To increment

© Mirza Ahad Baig, Danny Hendler, Alessia Milani, and Corentin Travers;
licensed under Creative Commons License CC-BY

33rd International Symposium on Distributed Computing (DISC 2019).
Editor: Jukka Suomela; Article No. 3; pp. 3:1–3:16

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

the counter, a process p simply increments its component. To read the counter's value, p invokes `Scan` and returns the sum of all components in the view it obtains. Since wait-free atomic snapshot can be implemented, using reads and writes only, in step complexity linear in the number of processes n [8, 14], so can counters.

Indeed, a well-known result by Jayanti, Tan and Toueg [16] proved a linear lower bound on the worst-case step complexity of obstruction-free read/write implementations of a large class of shared objects that includes counters. Aspnes, Attiya and Censor-Hillel [4] observed that the lower bound holds only when numerous operations are applied to the object and does not rule out the possibility of obtaining algorithms whose step complexity is sub-linear when the number of operations is bounded. Leveraging this observation, they presented constructions of several data structures for which operations' step complexity is polylogarithmic in n as long as the object's value is polynomial in n. Specifically, they presented a wait-free counter for which the step complexities of `Inc` and `Read` operations are $O\big(min(\log n \log v, n)\big)$ and $O\big(min(\log v, n)\big)$, respectively, where v is the object's current value. However, the worst-case and amortized step complexities of the counter algorithm of [4] deteriorate as the number of `Inc` operations increases. For executions in which the number of `Inc` operations is exponential in n, both the worst-case and the amortized step complexities become the same as those of the snapshot-based algorithm, that is, linear in n.

Our contribution. The lower bound of [16] leaves open the question of whether there exists a counter algorithm with sub-linear *amortized* step complexity. In this paper, we answer this question in the affirmative, by presenting the first wait-free read/write counter whose amortized step complexity is polylogarithmic. This is the first non-blocking read/write counter that provides sub-linear amortized step complexity in *executions of arbitrary length*. Our counter implementation is based on the counter algorithm presented in [4]. Their counter algorithm uses max registers, an object type they introduced and implemented. A *max register* r supports a `WriteMax`(r,v) operation that writes a non-negative integer v to r and a `ReadMax`(r) operation that returns the maximum value previously written to r.

We present a novel wait-free deterministic implementation of an unbounded max register and "plug it" into the counter algorithm of [4], thus obtaining a counter with $O(\log^2 n)$ amortized step complexity. Aspnes et al. also presented an unbounded max register, however the step complexities of both `ReadMax` and `WriteMax` operations in their algorithm are $O\big(min(\log v, n)\big)$, where v is the objects's current value. Thus, executions of arbitrary length can have linear amortized complexity. Aspnes and Censor-Hiller [6] presented an unbounded max register implementation for which every operation terminates in a constant number of steps with high probability, under the assumption that the max register's value does not grow too quickly. Our unbounded max algorithm makes a similar assumption. The max register algorithm of [6] is randomized since it relies on a randomized helping mechanism, whereas ours is deterministic.

Using information-theoretic arguments, Jayanti established a logarithmic lower bound on the *worst-case* operation step complexity for obstruction-free implementations of a set of one-time objects that includes a `fetch&increment` object, from operations such as load-linked/store-condition, move and swap [15]. Attiya and Hendler [9] presented lower bounds on the time and space complexities of obstruction-free implementations of several objects from k-word compare-and-swap operations. Specifically, using an information-theoretic argument as well, they proved a logarithmic lower bound on the *amortized* step complexity of implementing an obstruction-free one-time `fetch&increment` object [9, Theorem 9]. Their proof can be modified in a straightforward manner to establish the same result for counters, implying that our algorithm is optimal in terms of amortized step complexity up to a logarithmic factor.

The rest of this paper is organized as follows. We present the system model we assume and additional required definitions in Section 2. In Section 3, we present our key technical contribution – an unbounded max register algorithm that guarantees linearizability and logarithmic amortized step complexity when its value is not increased "too quickly". In Section 4, we prove that by "plugging" our unbounded max register into the counter algorithm of [4] (instead of using the max register algorithm of [4]) we obtain a linearizable counter with polylogarithmic amortized step complexity. The paper is concluded with a short discussion in Section 5.

2 Model and Preliminaries

Read/write shared memory. We consider a standard shared-memory model, where a set $\mathcal{P}$ of n crash-prone asynchronous processes communicate via shared *registers*, supporting only atomic read and write operations. A concurrent object *implementation* specifies the object's state representation and the algorithms processes follow when they perform operations supported by the object. An *execution* is a series of *steps* performed by processes as they follow their algorithms, in each of which a process applies at most a single read or write operation to a register (possibly in addition to some local computation). In what follows, we only consider finite executions. Roughly speaking, an implementation is *linearizable* [13] if each operation appears to take effect atomically at some point between its invocation and response; it is *wait-free* [11] if each process completes its operation if it performs a sufficiently large number of steps; it is *lock-free* if at least one process completes its operation after a sufficiently large number of steps is performed; it is *obstruction-free* [12] if each process completes its operation if it performs a sufficiently large number of steps when running solo. Operation Op_1 *precedes* operation Op_2 in an execution E, if Op_1's response appears in E before Op_2's invocation.

Complexity measure. The worst-case *amortized step complexity* (henceforth simply amortized step complexity) is defined as the worst-case (taken over all possible executions) average number of steps performed by operations. It measures the performance of an implementation as a whole rather than the performances of individual operations. Indeed, in an execution of a lock-free implementation, some operations may never terminate and the worst-case operation step complexity may thus be infinite. More precisely, given a finite execution E, an operation Op *appears* in E if it is invoked in E. We denote by $Nsteps(Op, E)$ the number of steps performed by Op in E and by $Ops(E)$ the set of operations that appear in E. The amortized step complexity of an implementation A is then:

$$AmtSteps(A) = \max_{E\text{: finite execution of } A} \frac{\sum_{Op \in Ops(E)} Nsteps(Op, E)}{|Ops(E)|}.$$

Max registers. A *max register* r supports a `WriteMax`(r,v) operation that writes a non-negative integer $v \geq 0$ to r and a `ReadMax`(r) operation that returns the maximum value previously written to r. A *bounded* max register $\texttt{MaxReg}_m$ can assume values from $\{0, \ldots, m-1\}$, for some integer m. An *unbounded* max register `UnboundedMaxReg` can store any non-negative integer.

3 Polylogarithmic Amortized Step Complexity Max Register

The pseudo-code of our unbounded max register is presented in Algorithm 1. Lines in black font constitute a lock-free version of the algorithm, which we describe and analyze in this section. Lines in lighter (metal) color add a helping mechanism that makes the algorithm wait-free. For presentation simplicity, we defer the description of this mechanism to Subsection 3.3. We proceed with a description of Algorithm 1. An $\texttt{UnboundedMaxReg}_m$ object M consists of an infinite number of shared bounded $\texttt{MaxReg}_m$ max registers, denoted $\texttt{max}_j$, for all $j \in \mathbb{N}_0$. Register $\texttt{max}_j$ will be used for representing values in the range $[m \cdot j, m \cdot (j+1)-1]$. Henceforth, the subscript m in the type $\texttt{UnboundedMaxReg}_m$ refers to the bound m of the bounded max registers used by objects of this type. Each bounded max register $\texttt{max}_j$ is associated with a shared $\texttt{switch}_j$ bit. All max registers and their corresponding switches are initialized to 0. Each process i has a variable $\texttt{last}_i$, storing the largest index j such that i accessed $\texttt{max}_j$, initialized to 0 as well.

The `Write` function. To write value v, process i first computes the index k of the bounded max register to write to and the residue v' to be written to it (lines 2–3). Next, i checks in line 4 whether $\texttt{max}_k$ is *obsolete*. We say that a (bounded) max register is obsolete, if its corresponding switch is set, indicating that values were already written to higher-indexed max registers and thus $\texttt{max}_k$ should no longer be accessed. If $\texttt{max}_k$ is obsolete, i does not need to write to it, so it proceeds to line 12 for increasing its *last* index, if required, and returns. Otherwise, $\texttt{max}_k$ is not obsolete, so i writes to it the residue v' (line 5). If the max object written to is not the first (line 6), then i ensures that the previous max object is obsolete (lines 8, 11), updates its *last* index (line 12), if required, and returns.

Algorithm 1 Unbounded Max Register $\texttt{UnboundedMaxReg}_m$, code for process i.

Shared variables:

- $\texttt{switch}_j \in \{0,1\}$: a 1-bit register for each $j \in \mathbb{N}_0$, initially all 0
- $\texttt{max}_j$: a $\texttt{MaxReg}_m$ object for each $j \in \mathbb{N}_0$, initially all 0
- $\texttt{last}_i \in \mathbb{N}_0$: stores the largest index j such that process i accessed $\texttt{max}_j$, initially 0
- $\texttt{H}[n][n]$ initially all 0 : a 2D integer array, $\texttt{H}[i][j]$ used by process j to help process i
- $\texttt{nextToHelp}_i$: identifer of last process helped by i

```
1:  function Write(UnboundedMaxReg_m, v)
2:      v' ← v mod m
3:      k ← ⌊v/m⌋
4:      if switch_k = 0 then
5:          WriteMax(max_k, v')
6:          if k > 0 then
7:              curMax ← ReadMax(max_{k-1}) + (k - 1) · m
8:              if switch_{k-1} = 0 then
9:                  H[nextToHelp_i][i] ← curMax
10:                 nextToHelp_i ← (nextToHelp_i + 1) mod n
11:                 switch_{k-1} ← 1
12:     last_i ← max(k, last_i)
13: function Read(UnboundedMaxReg_m)
14:     local c initially 0
15:     while switch_{last_i} ≠ 0 do
16:         last_i ← last_i + 1, c ← c + 1
17:         if (c mod n) = 0 then
18:             if (hVal ← GetHelp(c)) > 0 then return hVal
19:     v ← ReadMax(max_{last_i})
20:     return v + (last_i · m)
```

The `Read` function. Process i scans the switches in increasing order in lines 15–16, increasing the value of its `last` index in the process, until it finds the first non-obsolete bounded max register (this might never happen). Once it does, it reads the maximum residue previously written to that max object (line 19), adds to the residue a multiple of m corresponding to the index of that (non-obsolete) max register and returns the sum (line 20).

3.1 Linearizability

The correctness of Algorithm 1 is guaranteed only in executions in which the max register's value is increased in bounded increments. This requirement is formalized by the following definition.

▶ **Definition 1** (ℓ-Bounded-Increment Execution). *Let E be an execution and let M be an `UnboundedMaxReg` object. We say that E is an ℓ-bounded-increment execution for M if for each write operation $op = \texttt{Write}(v)$ on M in E, with $v > \ell$, there exists a write operation $op' = \texttt{Write}(v')$ on M in E that precedes op, such that $v - \ell \leq v' < v$.*

In Section 4, we present an n-process unbounded counter implementation that uses `UnboundedMaxReg` objects. As we prove, all the executions of that counter are n-bounded-increment executions for all these objects. Let M be an $\texttt{UnboundedMaxReg}_m$ object, implemented by Algorithm 1, for $m \geq n$, and let E be an n-bounded-increment execution for M, we now show that M is linearizable in E. We classify every write operation W on M that appears in E to exactly one of the 4 following types.

- **(i)** W did not yet execute line 4 in E.
- **(ii)** W executed line 4 and read $switch_k = 0$, but its `WriteMax` in line 5 was not yet linearized.
- **(iii)** W executed line 4, read $switch_k = 0$ and its `WriteMax` operation in line 5 was linearized. We say that W *is associated with* that `WriteMax` operation.
- **(iv)** W executed line 4 and read $switch_k = 1$.

Similarly, we classify every read operation R on M to the following 2 types:

- **(i)** R did not yet perform in E a `ReadMax` operation in line 19 that was linearized.
- **(ii)** R read $switch_k = 0$, for some k, and its `ReadMax` in line 19 was linearized. We say that R *is associated with* that `ReadMax` operation.

We associate with each $k \in \mathbb{N}_0$ two sets of operations on M in E, denoted $Down_k$ and $Futile_k$. Operations on M are partitioned into these sets as follows:

- $Down_k$ contains `Write` operations of type (iii) and `Read` operations of type (ii) that are associated with `WriteMax`/`ReadMax` operations on $\texttt{max}_k$.
- $Futile_k$ contains `Write` operations of type (iv).

Operations that were not assigned to any *Down* or *Futile* set are `Write` operations of types (i) and (ii) and `Read` operations of type (i). All these operations did not complete in E and will not appear in its linearization. We refer to these as *removed operations*. The rest of the operations are linearized according to the following *ordering rules*.

1. For all pairs k, k' such that $k < k'$, all the operations in $Down_k$ are ordered before all the operations in $Down_{k'}$.
2. We order the operations within each set $Down_k$ according to the linearization order of the `WriteMax` and `ReadMax` operations on the $\texttt{max}_k$ register with which they are associated.

3. Rules 1-2 order all *Down* operations. Enumerate them as Dop_1, $Dop_2, \ldots, Dop_r$. For any futile operation $Fop \in \bigcup_k Futile_k$, define the set

$$\mathcal{S}_{Fop} = \{Dop \in \cup_{k \in \mathbb{N}_0} Down_k \mid Fop \text{ precedes } Dop \text{ in } E\}.$$

If S_{Fop} is empty, we put *Fop* after Dop_r. Otherwise, let Dop_i be the least operation in S_{Fop} according to the ordering on *Down* operations, we put *Fop* immediately before Dop_i. For each set of the *Futile* operations put either immediately before some Dop_i or after Dop_r, the set of *Futile* operations is ordered according to their real-time order in E.[1]

Rules 1–3 define a full ordering among all non-removed operations.

▶ **Observation 2.** *An operation Op on M in E is associated with a `ReadMax` or `WriteMax` operation on `max`$_k$ if and only if $Op \in Down_k$.*

▶ **Observation 3.** *The sets $Down_k$ and* Futile$_k$ *(for all values of k) are mutually exclusive and contain all the operations on M that appeared in E except for removed operations.*

▷ Claim 4. M's $switch_j$ switches are set to 1 in E in increasing order, starting from $switch_0$.

Proof. Follows since M is an `UnboundedMaxReg`$_m$ object, for $m \geq n$, E is an n-bounded-increment execution for M, and from Lines 8,11. ◀

▷ Claim 5. For all $k' \leq k$, there are no two operations *Fop*, *Dop* such that $Fop \in Futile_k$, $Dop \in Down_{k'}$ and *Fop* is linearized before *Dop* in the ordering given by rules 1-3.

Proof. Suppose towards a contradiction that *Fop* is linearized before *Dop*. If *Fop* was placed after Dop_r when applying rule 3, we immediately reach a contradiction. Assume otherwise, then, from rule 3, there exists a *Down* operation Dop_i, such that *Fop* precedes Dop_i, *Fop* is linearized before Dop_i, and no *Down* operation is linearized between *Fop* and Dop_i. Consequently, it must be that *Dop* is linearized after Dop_i. From rules 1-2, we have that $Dop_i \in Down_{k_1}$ such that $k_1 \leq k' \leq k$. Since $Dop_i \in Down_{k_1}$, Dop_i reads 0 from $Switch_{k_1}$. However, *Fop* reads $Switch_k = 1$ before Dop_i starts, hence, by Claim 4, $Switch_{k_1} = 1$ when Dop_i starts. This is a contradiction. ◀

▶ **Lemma 6.** *Ordering rules 1-3 define a sequential order between E's non-removed operations that preserves the real-time order between non-overlapping operations in E.*

Proof. From ordering Rule 2, Observation 2 and the linearizability of the max_j objects, for each j, the real-time order between all operations in $Down_k$ is preserved. From Claim 4, M's switches are set to 1 in increasing order. Consequently, for any two operations $Op \in Down_k$ and $Op' \in Down_{k'}$, such that $k < k'$, Op' does not precede Op in E. It follows that the real-time order between each pair of *Down* operations is preserved by the linearization.

It remains to argue about *Futile* operations. Let $Dop_1, Dop_2, \ldots, Dop_r$ be the linear order among all *Down* operations, as specified by rules 1-2. Let Fop_1, Fop_2 be *Futile* operations such that Fop_1 is linearized before Fop_2. There are two cases to consider. If both operations are put after Dop_r or immediately before the same operation Dop_i then, according to rule 3, their order preserves E's real-time order. Otherwise, there exists at least one *Down* operation linearized between them. Let *Dop* be the first *Down* operation ordered

[1] In general, this induces a partial order on *Futile* operations, which can be extended to a full order arbitrarily.

after Fop_1. From rule 3, Fop_1 precedes Dop in real-time order. Suppose Fop_2 precedes Fop_1 in real-time order, then $Dop \in \mathcal{S}_{Fop_2}$ holds. Since Fop_2 is linearized by rule 3 before all the *Down* operations that follow it in real-time order, this is a contradiction.

Let Fop and Dop respectively be a $Futile_k$ and a $Down_{k'}$ operation. Suppose Fop is linearized before Dop but Dop precedes Fop in real-time order. If $k' \leq k$, then, by Claim 5, this is a contradiction. Assume, then, that $k < k'$ holds and consider the application of ordering rule 3 to Fop. If Fop is put after Dop_r, then it is linearized after Dop, which is a contradiction. Assume, then, that Fop is put by rule 3 immediately before some $Dop_i \in \mathcal{S}_{Fop}$, so Fop precedes Dop_i in real-time order. It follows that Dop precedes Dop_i in real-time order, so Dop is linearized before Dop_i. Since no *Down* operation can be linearized between Fop and Dop_i, it follows that Dop is linearized before Fop. This is a contradiction.

Finally, suppose that Fop precedes Dop in real-time order. In this case, from rule 3, Fop is linearized before Dop, preserving real-time order. ◀

▶ **Lemma 7.** *The linearization defined by ordering rules 1-3 satisfies the sequential semantics of a max register.*

Proof. For an integer $v \in \mathbb{N}_0$, let $v' = v \mod m$ and $k = \lfloor v/m \rfloor$. Consider a `Read` operation Op_r on M in E that returns value v. Then Op_r is associated with a `ReadMax`($\mathtt{max}_k$) operation that returned v'. First, we prove that there is a `Write` operation Op_w that wrote v to M and Op_w is linearized before Op_r. From the linearizability of $\mathtt{max}_k$, there is a `WriteMax`($\mathtt{max}_k, v'$) operation that is linearized before the `ReadMax`($\mathtt{max}_k$) associated with Op_r. Thus, there is a `Write` operation $Op_w(v' + k \cdot m)$ on M and, by Observation 2, it belongs to $Down_k$, so Op_w is ordered before Op_r according to ordering rule 2. To conclude the proof, we show that there is no `Write` operation Op_1 that writes value $v_1 > v$ to M and is linearized before Op_r. Suppose towards a contradiction that Op_1 exists. The following two cases exist:

- $\lfloor v_1/m \rfloor = k$. This implies that $(v_1 \mod m) > (v \mod m)$. If $Op_1 \in Down_k$, this contradicts the linearizability of $\mathtt{max}_k$, because the `ReadMax` operation associated with Op_r does not return the maximum value written to $\mathtt{max}_k$ before it. If $Op_1 \in Futile_k$, this contradicts Claim 5.
- $\lfloor v_1/m \rfloor > k$. In this case, either $Op_1 \in Down_{k'}$ or $Op_2 \in Futile_{k'}$, for some $k' > k$. In the first case, Op_1 is linearized after Op_r by rule 1. In the second case, Claim 5 ensures that Op_1 is linearized after Op_r. ◀

▶ **Lemma 8.** *Algorithm 1 (without the helping mechanism) is lock-free.*

Proof. `Write` operations perform a single invocation of the wait-free `WriteMax` operation and a constant number of additional steps, hence they are wait-free. A `Read` operation may loop forever in lines 15–16, searching for a non-obsolete max register, but only if `Write` operations keep making additional max registers obsolete (in line 11). If no more `Write` operations complete, each `Read` operation is guaranteed to complete. ◀

3.2 Step Complexity Analysis

The step complexity analysis provided in this section relates to the implementation of Algorithm 1 without the helping mechanism. In the following, we denote by $Ops(E)$ the set of all operations that appear in E and by $Ops_R(E)$ (resp. $Ops_W(E)$) the set of all `Read` operations (resp. all `Write` operations) that appear in E. For an operation Op, we let $Nsteps(Op, E)$ denote the number of steps performed by Op in E.

▶ **Lemma 9.** *If $m \geq n^2$, then the $\mathtt{UnboundedMaxReg}_m$ implementation of Algorithm 1 has amortized step complexity of $O(\log m)$ in any n-bounded-increment execution.*

Proof. Let E be an n-bounded-increment execution. We wish to bound:

$$AmtSteps(E) = \frac{\sum_{op \in Ops(E)} Nsteps(op, E)}{|Ops(E)|}. \tag{1}$$

Let r be the number of read operations and w be the number of write operations in $Ops(E)$. `WriteMax` and `ReadMax` operations on an m-bounded max register perform $O(\log m)$ steps each. Clearly from the pseudo-code of Algorithm 1, each `Write` operation performs a constant number of steps in addition to possibly invoking a single `WriteMax` operation, thus the step complexity of each `Write` operation is $O(\log m)$.

A `Read` operation Op performs $loop_{Op} + O(\log m)$ steps, where $loop_{Op}$ is the number of steps performed in the while loop of lines 15–16 and $O(\log m)$ is the number of steps performed by the invocation of `ReadMax` in line 19. We get:

$$AmtSteps(E) = O\Big(\big(\sum_{op \in Ops_W(E)} \log m + \sum_{op \in Ops_R(E)} \log m + loop_{op}\big)/(w+r)\Big). \tag{2}$$

If $r = 0$, then clearly $AmtSteps(E) = O(\log m)$, so assume that $r > 0$. From lines 12 and 16, for every process i, $last_i$ is never decreased and is incremented once in every iteration of the while loop of lines 15–16. Therefore:

$$\sum_{op \in Ops_R(E)} loop_{op} = O\Big(r + \sum_{i \in \mathcal{P}} last_i\Big). \tag{3}$$

Consequently,

$$AmtSteps(E) = O\Big(\frac{w \cdot \log m + r \cdot \log m + (r + \sum_{i \in \mathcal{P}} last_i)}{w + r}\Big). \tag{4}$$

Assume that max register $\mathtt{max}_\alpha$ is accessed in E. Since E is an n-bounded-increment execution and all $\mathtt{max}_j$ registers are m-bounded, at least $m \cdot (\alpha - 1)/n$ $Write$ operations have completed prior to this access. Letting $\mathcal{L} = \max_{i \in \mathcal{P}} last_i$ denote the maximum value of all $last_i$ variables at the end of E, we get that $w \geq m \cdot (\mathcal{L} - 1)/n$. Furthermore, $\sum_{i \in \mathcal{P}} last_i \leq n \cdot \mathcal{L}$. Thus,

$$\begin{aligned} AmtSteps(E) &= O\Big(\frac{w \log m + r \log m + (r + n \cdot \mathcal{L})}{w + r}\Big) = O\Big(\frac{(w+r)\log m}{w+r} + \frac{r}{w+r} + \frac{n \cdot \mathcal{L}}{w+r}\Big) \\ &= O\Big(\log m + \frac{n \cdot \mathcal{L}}{\frac{m}{n}(\mathcal{L} - 1) + r}\Big) = O\Big(\log m + \frac{\frac{n^2}{m}\mathcal{L}}{(\mathcal{L} - 1) + \frac{n}{m} r}\Big). \end{aligned} \tag{5}$$

The lemma now follows, since $r > 0$ and $m \geq n^2$ hold. ◀

From Lemmata 6–9, we obtain:

▶ **Theorem 10.** *Algorithm 1 is a linearizable implementation of an unbounded max register with amortized step complexity of $O(\log m)$ in any n-bounded-increment execution, if $m \geq n^2$. The algorithm (without the helping mechanism) is lock-free.*

Algorithm 2 The `GetHelp` utility function, code for process i.

Shared variables:
  $\texttt{HR}_i[n]$: an integer array, to which the i'th row in the H array is copied
  $\texttt{C}_i[n]$: an integer array, counting number of writes by each helper for process i

1: **function** GetHelp(c)
2:   **if** $c = n$ **then**
3:     **for** $j \in \{0, \ldots, n-1\}$ **do**
4:       $\texttt{HR}_i[j] \leftarrow \texttt{H}[i][j], \texttt{C}_i[j] \leftarrow 0$
5:   **else**
6:     **for** $j \in \{0, \ldots, n-1\}$ **do**
7:       **if** $\texttt{HR}_i[j] < \texttt{H}[i][j]$ **then**
8:         $\texttt{HR}_i[j] \leftarrow \texttt{H}[i][j]$, $\texttt{C}_i[j]++$
9:         **if** $\texttt{C}_i[j] = 2$ **then return** $\texttt{HR}_i[j]$
10:   **return** 0

3.3 The Helping Mechanism

We now explain the helping mechanism that makes Algorithm 1 wait-free (presented in the metal-colored lines of that algorithm). It uses a 2-dimensional shared array H. Entry $\texttt{H}[i][j]$ is used by process j to help process i by writing to it a (maximum) value of M that process j was able to compute. Each process i owns variable $\texttt{nextToHelp}_i$, storing the index of the next process it should help. Helping is attempted by process i inside `Write` operations, whenever i is about to make another max register obsolete. Specifically, if i is about to write to a max register $k > 0$ (line 6), it reads the maximum residue written so far to $\texttt{max}_{k-1}$, computes the corresponding value of M based on it and stores it to a local variable *curMax* (line 7). If $\texttt{switch}_{k-1}$ is 0 (line 8), then $\texttt{max}_{k-1}$ must be made obsolete. As we prove, in this case, *curMax* was indeed a value of M at some point during the execution interval of i'th `Write` operation, so i attempts to help process $\texttt{nextToHelp}_i$ by writing to the appropriate entry of array H and increments $\texttt{nextToHelp}_i$ modulo n (lines 9–10).

The goal of the helping mechanism is to ensure that every `Read` operation eventually completes. Every n iterations of the while loop of lines 15–18, the `GetHelp` utility function is called, receiving an integer that is a multiple of n, indicating whether or not this is its first invocation by the current `Read` operation (line 14, lines 17–18). If `GetHelp` returns a positive value then, as we prove, this was indeed M's value at some point during the execution interval of `Read`, so it returns this value in line 18. Otherwise, the search for a non-obsolete max register is resumed.

The pseudo-code of `GetHelp` is presented by Algorithm 2, described next. In its first invocation by `Read` operation R (performed by some process i), initialization is done by copying the i'th row of the H array to array $\texttt{HR}_i$ and initializing all elements of a second array $\texttt{C}_i$ to 0 (lines 2–4). Both $\texttt{HR}_i$ and $\texttt{C}_i$ are only accessed by process i. Element $\texttt{C}_i[j]$ counts the number of times in which i observed that it was helped by process j in the course of R. In the first invocation, 0 is returned (line 10), indicating that a maximum value is not yet available. In each subsequent invocation of `GetHelp` (lines 5–9), if any, i checks, for each j, if it was helped by j since the last time it read $\texttt{H}[i][j]$, in which case it updates $\texttt{HR}_i[j]$ and increments $\texttt{C}_i[j]$. If i was helped by some process j at least twice since R started then, as we prove, the maximum value computed by j for i was indeed M's value at some point during R's execution interval, so `GetHelp` returns it in line 9 and R then returns this value in line 18 of Algorithm 1. Otherwise, 0 is returned in line 10.

3.4 Correctness

In this section we prove that the algorithm with the helping mechanism (henceforth *the full algorithm*) is linearizable. We classify read and write operations to types as we did in Section 3.1, except that now we have a 3'rd class of read operations – those that return in line 18 of Algorithm 1 after being helped. We say that these are `Read` operations of type (iii).

Let R be a type (iii) `Read` operation by process i that returns value u and let $k' = \lfloor u/m \rfloor$, then there is a `Write` operation W by process j, concurrent with R, that wrote u to $\texttt{H}[i][j]$ (in Line 9 of Algorithm 1) after performing a `ReadMax` operation on $max_{k'}$ (in Line 7 of Algorithm 1) and R returns value u after reading it from $\texttt{H}[i][j]$ (Lines 8, 9 of `GetHelp`). We say that R *is associated with* that `ReadMax` operation.

As in Section 3.1, we partition the operations of E to the sets $Down_k$ and $Futile_k$, except that we now add each `Read` operation of type (iii) that is associated with a `ReadMax` on $\texttt{max}_k$ to $Down_k$. We use the ordering rules defined in Section 3.1 to linearize all of E's non-removed operations. It is easily verified that Observations 2–3 and Claim 4 hold also with the extended definition of the sets $Down_k$.

▶ **Observation 11.** *Let R be a type (iii) `Read` operation associated with a `ReadMax` operation R' on $max_{k'}$. Then all throughout the execution of R', $\texttt{switch}_{k'} = 0$ holds.*

Proof. Immediate from Claim 4 and the fact that the `Write` operation that invokes R' in line 7 of Algorithm 1 writes the value read by R' (in line 9) to the `H` array only after verifying that $\texttt{switch}_{k'} = 0$ holds (in line 8). ◀

Based on Observation 11, we now prove that Claim 5 holds for full algorithm.

▷ Claim 12. In any ordering of operations for the full algorithm, for all $k' \leq k$, there are no two operations *Fop*, *Dop* such that $Fop \in Futile_k$, $Dop \in Down_{k'}$ and *Fop* is linearized before *Dop* in the ordering given by rules 1–3.

Proof. Suppose towards a contradiction that *Fop* is linearized before *Dop*. If *Fop* was placed after Dop_r when applying rule 3, then we immediately reach a contradiction. Assume otherwise, then, from rule 3, there exists a *Down* operation Dop_i, such that *Fop* precedes Dop_i, *Fop* is linearized before Dop_i, and no *Down* operation is linearized between *Fop* and Dop_i. Consequently, it must be that *Dop* is linearized after Dop_i. From rules 1-2, we have that $Dop_i \in Down_{k_1}$ such that $k_1 \leq k' \leq k$. Since $Dop_i \in Down_{k_1}$, either Dop_i reads 0 from $Switch_{k_1}$ or, otherwise, it is a type (iii) `Read` operation, in which case, from Observation 11, $Switch_{k_1} = 0$ holds at some point during its execution. However, *Fop* reads $Switch_k = 1$ before Dop_i starts, hence, by Claim 4, $Switch_{k_1} = 1$ when Dop_i starts. This is a contradiction. ◁

We next show that Lemma 6 holds also for the full algorithm.

▶ **Lemma 13.** *Ordering rules 1-3 for the full algorithm define a sequential order between E's non-removed operations that preserves the real-time order between non-overlapping operations in E.*

Proof. In Lemma 6, the corresponding claim was proven w.r.t. the algorithm without the helping mechanism for operations of all types, except for `Read` operations of type (iii). A type (iii) operation $R \in Down_k$ by process i is associated with a `ReadMax` operation R' on $\texttt{max}_k$ invoked from a concurrent `Write` operation, performed by some process $j \neq i$. Since the condition of line 9 of `GetHelp` was satisfied when evaluated by i, the execution interval of R'

is fully contained within that of R. It follows that R can be linearized when R' is linearized on $\texttt{max}_k$. Thus, R is ordered w.r.t. other operations in $Down_k$ by applying ordering rule 2 to R' breaking ties arbitrarily which, from Observation 2 and the linearizability of max_k, ensures that real-time order is maintained between all operations in $Down_k$.

Let $Op \in Down_k$ and $Op' \in Down_{k'}$ be two operations such that $k < k'$. From Claim 4, M's switches are set to 1 in increasing order. Based on this and on Observation 11 (which is required if either Op or Op' is a type (iii) `Read` operation), Op' does not precede Op in E. it follows that the real-time order between each pair of $Down$ operations is preserved by the linearization.

Let Fop and Dop respectively be a $Futile_k$ and a $Down_{k'}$ operation. Suppose Fop is linearized before Dop but Dop precedes Fop in real-time order. If $k' \leq k$, then, by Claim 12, this is a contradiction. The rest of the proof proceeds exactly as in the proof of Lemma 6. ◀

It is easily verified that Lemma 7 holds also for the full algorithm. The only change required in its proof is to use Claim 12 instead of Claim 5.

▷ Claim 14. If a monotonically-increasing sequence of values is written to M, then some process performs line 9 of Algorithm 1 infinitely often.

Proof. If a monotonically-increasing sequence of values is written to M, then `max` registers are made obsolete infinitely often. Since a `max` register is only made obsolete in line 11 of Algorithm 1, it is immediate from the code that line 9 of that algorithm is performed infinitely often as well. Since the number of processes is finite, it follows that some process performs that line infinitely often. ◁

▶ **Lemma 15.** *The full Algorithm 1 is wait-free.*

Proof. As proven in Lemma 8, the algorithm is lock-free and `Write` operations are wait-free. It remains to show that `Read` operations are wait-free as well. From Claim 14, if a monotonically-increasing sequence of values is written to M, then there is some process j that performs line 9 of Algorithm 1 infinitely often. Thus, any `Read` operation, say by process i, eventually either finds a non-obsolete `max` register in line 15 or increments $\texttt{C}_i[j]$ twice in line 9 of `GetHelp` and is therefore able to terminate.

Otherwise, there is no such sequence of monotonically-increasing values. Thus, starting from some point in the execution, M's value does not increase, so the set of obsolete `max` object stops growing, hence every `Read` operation that does not fail-stop eventually reaches a non-obsolete `max` register and completes. ◀

▶ **Theorem 16.** *If $m \geq n^2$, then the full algorithm is a wait-free linearizable n-process implementation of an unbounded max register with amortized step complexity of $O(\log m)$ in any n-bounded-increment execution.*

Proof. From Lemmata 7, 13 and 15 the full algorithm is linearizable and wait-free, so it remains to argue regarding its complexity. In Algorithm 2, every iteration of the `for` loop at either line 3 or line 6 incurs a constant number of steps. Thus, every invocation of `GetHelp` incurs $O(n)$ steps. In Algorithm 1, a `Write` operation performs at most one `WriteMax` and at most one `ReadMax` operation, incurring a total of $O(\log m)$ steps. We note that any `Read` operation invokes `GetHelp` once every $k \cdot n$ steps, for some $k > 1$, when $c = 0 \mod n$. Thus, at any point in the course of the execution, the number of steps taken by a `Read` operation R inside `GetHelp` is $O(loop_R)$. Consequently, as in the proof of Lemma 9, we get:

$$AmtSteps(E) = O\Big(\big(\sum_{op \in Ops_W(E)} \log m + \sum_{op \in Ops_R(E)} \log m + loop_{op}\big)/(w+r)\Big) = O(\log m). \quad (6)$$

◀

Algorithm 3 An n-process counter C_j, code for process i.

```
Shared variables:
      R: an n-process UnboundedMaxReg_{n^2} object, initially 0
      If j > 1: left: a C_{⌈j/2⌉} counter object, initially 0
               right: a C_{j-⌈j/2⌉} counter object, initially 0
 1: function Inc(C_j)
 2:     if j = 1 then
 3:         v ← ReadMax(R)
 4:         WriteMax(R, v + 1)
 5:     else
 6:         if i's C_1 leaf-counter is on the left sub-tree then Inc(left) else Inc(right)
 7:         v_0 ← read(left)
 8:         v_1 ← read(right)
 9:         WriteMax(R, v_0 + v_1)
10: function Read(C_j)
11:     return ReadMax(R)
```

4 Wait-Free Counter with Polylogarithmic Amortized Step Complexity

Algorithm 3 presents a wait-free recursive construction of a linearizable counter that has polylogarithmic amortized step complexity in all executions, regardless of their length. The algorithm is essentially the same as the (non-recursive) counter construction of Aspnes et al. [4], except that the latter uses the max registers of [4], whose amortized step complexity is linear for sufficiently long executions, whereas ours uses our wait-free unbounded max registers.

Let C_j denote a counter, shared by n processes, implemented by Algorithm 3. For simplicity and without loss of generality, assume in the following that each of n and j is an integral power of 2. C_j's value is stored in an n-process wait-free unbounded max register R, which is of type $\texttt{UnboundedMaxReg}_{n^2}$. If $j > 1$ holds, then C_j also contains two $C_{j/2}$ child-counters – `left` and `right`. A counter C_n serves as a root of a tree of counters and all processes can invoke `Inc` operations on C_n. At the bottom layer of the tree, each process i is associated with a single C_1 leaf-counter on which only i can invoke `Inc` operations.

To read C_j, process i simply invokes a `ReadMax` operation on C_j's R object and returns the response (line 11). Incrementing a C_1 object consists of simply reading R and writing to it a value larger by one (lines 3–4). To increment a C_j counter, for $j > 1$, process i increments either the `left` or the `right` child counter, depending on whether its C_1 leaf-counter is on the left or the right subtree of C_j, reads the values of both child counters and writes their sum to R (lines 6–9). Observe that at most j distinct processes can invoke `Inc` operations on any specific C_j counter.

In the following proofs we let $\mathcal{C}$ denote a C_n object implemented by Algorithm 3 and E be an execution of $\mathcal{C}$.

▶ **Lemma 17.** *The C_j counter implementation of Algorithm 3 is linearizable.*

Proof. The proof is by induction on j.

Base Case. For $j = 1$, the `UnboundedMaxReg` object R of a C_1 counter may only be incremented by a single process. Since R's value is always increased by exactly 1, the execution is 1-bounded-increment for R, so the correctness of R follows from Theorem 16. Increment operations on C_1 are linearized when the `WriteMax` operation invoked in line 4 is linearized and read operations on C_1 are linearized when the `ReadMax` operation invoked in line 11 is linearized.

Induction Hypothesis. For all $k < j$, C_k is a linearizable counter and the value of the max object R it uses is never increased by more than k.

Inductive Step.

▶ **Sub-Lemma 17.1.** *E is a j-bounded-increment execution for $C_j.R$.*

Proof. The proof is divided into two parts. We first prove the left-hand inequality of Definition 1. Let E' be a prefix of E immediately after which process p is about to invoke a `WriteMax()` operation Op_v on $C_j.R$ with input v (in line 9). Let $\mathcal{I}$ be the set of `Inc` operations that have completed on C_j in E'. Observe that each operation $Op \in \mathcal{I}$ has performed one `Inc` operation on either C_j.`left` or C_j.`right`. We partition $\mathcal{I}$ accordingly: $\mathcal{I} = \mathcal{I}_0 \cup \mathcal{I}_1$, where for any $Op \in \mathcal{I}$, $Op \in \mathcal{I}_0$ if Op performed an `Inc` operation on C_j.`left` and $Op \in \mathcal{I}_1$ if Op performed an `Inc` operation on C_j.`right`.

By IH, both C_j.`left` and C_j.`right` are linearizable counters. Let $Op_0 \in \mathcal{I}_0$ be the operation whose `Inc` operation on C_j.`left` is linearized last among all `Inc` operations on C_j.`left` performed by the operations in $\mathcal{I}_0$. Let c_0 be the value of C_j.`left` immediately after the `Inc` operation on that object by Op_0. Op_1 and c_1 are defined similarly. From lines 7–9, for each $r \in \{0, 1\}$, after performing an `Inc` operation on either C_j.`left` or C_j.`right`, Op_r performs *read* operations on both C_j.`left` and C_j.`right` before writing the sum u_r of the values read to $C_j.R$. We show that $v' = \max\{u_0, u_1\} \geq c_0 + c_1$. Indeed, assume that Op_0's *read* operation on $C_j.right$ returns a value strictly smaller than c_1. Then, Op_1's `Inc` operation on C_j.`right` is linearized after Op_0's *Read* operation on C_j.`right`. It thus follows that Op_1's *read* operation on C_j.`left` starts after Op_0's `Inc` operation on C_j.`left` has completed. We thus conclude that $u_1 \geq c_0 + c_1$.

As both Op_0 and Op_1 have completed in E', a `WriteMax` operation on R of value $v' \geq c_0 + c_1$ has completed in E'. If $v \leq v'$ then $v - j \leq v'$ and the claim holds. Otherwise, again from lines 7–9, the operand v of the `WriteMax` operation Op_v is the sum of the values v_0, v_1 returned by the `Read` operations performed on the counters C_j.`left` and C_j.`right`, respectively. $v_0 = c_0 + \delta$, for $\delta > 0$, implies that there are δ `Inc` operations on C_j.`left` that have been linearized after the `Inc` operation on the same counter by Op_0. From the definition of Op_0, these δ operations take place within δ `Inc` operations on C_j that did not complete in E'. The same argument applies for v_1. Since there are at most j processes that may invoke `Inc` operations on C_j and thus at most j incomplete `Inc` operations on C_j after E', it follows that $v = v_0 + v_1 \leq j + c_0 + c_1$. Hence, there is a value $v' = \max\{u_0, u_1\}$ such that $v - v' \leq j$ and a `WriteMax`(v') on R has completed before the operation $Op_v =$ `WriteMax`(v) on R starts.

We next prove both inequalities of Definition 1. Let Op be a `WriteMax` operation on $C_j.R$ with input $v > j$. The first part above established that there exists a `WriteMax` operation Op' on $C_j.R$ with input v' that finishes before Op starts, such that $v - n \leq v'$. Assume that $v' \geq v$. Let $\mathcal{O}_>$ be the set of `WriteMax` operations on $C_j.R$ that (1) precede Op and (2) whose input is larger than or equal to v. We define a partial order $\prec$ on the operations in $\mathcal{O}_>$ as follows:

$$\forall W, W' \in \mathcal{O}_>, W \prec W' \iff W \text{ precedes } W' \text{ in } E.$$

Let us observe that $\mathcal{O}_>$ is non-empty and finite. The latter is because E is finite and so only finitely many operation precede Op in E and the former follows from the existence of Op'. Consider any minimal element in the partially ordered set $\mathcal{O}_>$, that is any operation W such that for any operation $W' \in \mathcal{O}_>$, W' does not precede W. Since $\mathcal{O}_>$ is finite, there is

at least one such operation W. Let in_W denote its input. Since $W \in \mathcal{O}_>$, we have $in_W \geq v$. Also, by applying the left-hand inequality (proved in the first part of the proof) to W, there exists an operation W' with input $in_{W'}$ that precedes W such that $in_{W'} \geq in_W - j \geq v - j$. As $W' \prec W$, and W is chosen as a minimal element of $\mathcal{O}_>$, it follows that $W' \notin \mathcal{O}_>$. Since W' precedes both W and Op, we get that $in_{W'} < v$, which concludes the proof. ◀

From Sub-lemma 17.1 and Theorem 16 we conclude that $C_n.R$ is linearizable in E. Based on this, the proof proceeds similarly to the proof of [4, Lemma 4].

From IH, C_j.`left` and C_j.`right` are linearizable counters. We associate with every increment operation Op on C_j a value as follows. Let c_0 and c_1 respectively denote the values of C_j.`left` and C_j.`right` immediately after p's increment of C_j's child (corresponding to p's identifer), in line 6, is linearized. Then we associate with Op the value $v = c_0 + c_1$. We linearize an *Inc* operation Op, associated with value v, when a value $v' \geq v$ is first written to $C_j.R$ in line 9 (either by p or by another process). We linearize a *Read* operation on C_j when it reads $C_j.R$ in line 11.

We now prove that each linearization point lies within its operation execution interval. Consider an `Inc` operation Op associated with value v. A value $v' \geq v$ cannot be written to $C_j.R$ before Op starts, because, from the linearizability of C_j.`left` and C_j.`right`, before Op starts, the sum of these two counters is less than $c_0 + c_1$. Since Op itself writes value v to $C_j.R$ before it terminates, the linearization point occurs before Op terminates. The fact that the linearization point of a *Read* operation on C_j lies within its execution interval follows immediately from the linearizability of $C_j.R$, established by Sub-lemma 17.1. Finally, the linearization points result in a valid sequential execution, because every *Read* operation on C_j that returns value v is preceded by exactly v `Inc` operations on C_j. ◀

▶ **Lemma 18.** *Algorithm 3 has $O(\log^2 n)$ amortized operation step complexity.*

Proof. From Algorithm 3 and the fact that $\mathcal{C}$ is shared by n processes, every operation on $\mathcal{C}$ applies a constant number of `ReadMax`/`WriteMax` operations to each of $O(\log n)$ different `UnboundedMaxReg`$_{n^2}$ objects, as the recursive calls in lines 7–9 and 11 unfold. Letting $COps(E)$ denote the number of operations on $\mathcal{C}$ that appear in E, the total number of `ReadMax`/`WriteMax` operations on all the implementation's `UnboundedMaxReg`$_{n^2}$ objects is therefore $O\big(\log n \cdot COps(E)\big)$. From Theorem 16, letting $m = n^2$, it follows that the total number of steps performed in E is $O\big(\log^2 n \cdot COps(E)\big)$. ◀

▶ **Theorem 19.** *Algorithm 3 is a wait-free linearizable n-process implementation of an unbounded counter with amortized step complexity of $O(\log^2 n)$.*

Proof. From Lemma 17, the algorithm is linearizable. From Lemma 15, all the `UnboundedMaxReg` objects used by Algorithm 3 are wait-free, thus, clearly from the pseudo-code, Algorithm 3 is wait-free as well. The claimed complexity follows from Lemma 18. ◀

Attiya and Hendler proved a logarithmic lower bound on the amortized step complexity of implementing an obstruction-free one-time `fetch&increment` object from read, write and k-word compare-and-swap operations [9, Theorem 9]. Their proof can be easily adapted to obtain the following result:

▶ **Lemma 20.** *Any n-process obstruction-free implementation from read/write registers of a counter object has an execution that contains $\Omega(n \log n)$ steps, in which every process performs a single `Inc` operation followed by a single `Read` operation.*

Lemma 20 establishes that every non-blocking read/write counter implementation has an execution whose amortized step complexity is at least logarithmic in the number of processes, showing that our counter algorithm is optimal in terms of amortized step complexity up to a logarithmic factor.

5 Discussion

In this work, we presented the first non-blocking read/write counter algorithm that provides sub-linear amortized step complexity in all executions, regardless of their length. The amortized operation step complexity of our algorithm is $O(\log^2 n)$, where n is the number of processes sharing the implementation. This is optimal up to a logarithmic factor, since there exists a logarithmic lower bound on the amortized step complexity of n-process one-time counters.

It is unclear whether there exists a wait-free (or even lock-free or obstruction-free) read/write counter implementation with $o(\log^2 n)$ amortized step complexity. Interestingly, a similar gap between an $O(\log^2 n)$ upper bound and an $\Omega(\log n)$ lower bound exists for the *worst-case* step complexity of counters [4].

The space complexity of our counter is infinite, since it uses our unbounded max registers, and each of these encapsulates an infinite number of bounded max registers. A second question is that of finding a bounded-space read/write counter with sub-linear amortized step complexity. These questions are left for future work.

References

1. Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM (JACM)*, 40(4):873–890, 1993.
2. Yehuda Afek, Haim Kaplan, Boris Korenfeld, Adam Morrison, and Robert E Tarjan. The CB tree: a practical concurrent self-adjusting search tree. *Distributed computing*, 27(6):393–417, 2014.
3. James Anderson. Composite registers. *Distributed Computing*, 6(3):141–154, 1993.
4. James Aspnes, Hagit Attiya, and Keren Censor-Hillel. Polylogarithmic concurrent data structures from monotone circuits. *J. ACM*, 59(1):2:1–2:24, 2012.
5. James Aspnes and Keren Censor. Approximate shared-memory counting despite a strong adversary. *ACM Trans. Algorithms*, 6(2):25:1–25:23, 2010.
6. James Aspnes and Keren Censor-Hillel. Atomic Snapshots in $O(\log^3 n)$ Steps Using Randomized Helping. In *27th International Symposium on Distributed Computing (DISC)*, volume 8205 of *Lecture Notes in Computer Science*, pages 254–268. Springer, 2013.
7. James Aspnes and Maurice Herlihy. Fast randomized consensus using shared memory. *Journal of algorithms*, 11(3):441–461, 1990.
8. Hagit Attiya and Arie Fouren. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM Journal on Computing*, 31(2):642–664, 2001.
9. Hagit Attiya and Danny Hendler. Time and Space Lower Bounds for Implementations Using k-CAS. *IEEE Trans. Parallel Distrib. Syst.*, 21(2):162–173, 2010.
10. Michael A. Bender and Seth Gilbert. Mutual Exclusion with $O(\log^2 \log n)$ Amortized Work. In *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, pages 728–737. IEEE, 2011.
11. Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.
12. Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-Free Synchronization: Double-Ended Queues as an Example. In *23rd International Conference on Distributed Computing Systems (ICDCS)*, pages 522–529. IEEE Computer Society, 2003.

13 Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

14 Michiko Inoue, Toshimitsu Masuzawa, Wei Chen, and Nobuki Tokura. Linear-time snapshot using multi-writer multi-reader registers. In *International Workshop on Distributed Algorithms*, pages 130–140. Springer, 1994.

15 Prasad Jayanti. A Time Complexity Lower Bound for Randomized Implementations of Some Shared Objects. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, PODC '98*, pages 201–210, 1998.

16 Prasad Jayanti, King Tan, and Sam Toueg. Time and Space Lower Bounds for Nonblocking Implementations. *SIAM J. Comput.*, 30(2), 2000.

17 Shlomo Moran and Gadi Taubenfeld. A lower bound on wait-free counting. *J. Algorithms*, 24(1):1–19, 1997.

18 Shlomo Moran, Gadi Taubenfeld, and Irit Yadin. Concurrent Counting. *J. Comput. Syst. Sci.*, 53(1):61–78, 1996.

Distributed Algorithms for Low Stretch Spanning Trees

Ruben Becker
Gran Sasso Science Institute, L'Aquila, Italy
ruben.becker@gssi.it

Yuval Emek
Technion – Israel Institute of Technology, Haifa, Israel
yemek@technion.ac.il

Mohsen Ghaffari
ETH Zürich, Switzerland
ghaffari@inf.ethz.ch

Christoph Lenzen
MPI for Informatics, Saarland Informatics Campus, Saarbrücken, Germany
clenzen@mpi-inf.mpg.de

Abstract

Given an undirected graph with integer edge lengths, we study the problem of approximating the distances in the graph by a spanning tree based on the notion of *stretch*. Our main contribution is a distributed algorithm in the CONGEST model of computation that constructs a random spanning tree with the guarantee that the expected stretch of every edge is $O(\log^3 n)$, where n is the number of nodes in the graph. If the graph is unweighted, then this algorithm can be implemented to run in $O(D)$ rounds, where D is the hop-diameter of the graph, thus being asymptotically optimal. In the weighted case, the run-time of our algorithm matches the currently best known bound for exact distance computations, i.e., $\tilde{O}(\min\{\sqrt{nD}, \sqrt{n}D^{1/4} + n^{3/5} + D\})$. We stress that this is the first distributed construction of spanning trees leading to poly-logarithmic expected stretch with non-trivial running time.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms; Theory of computation → Distributed algorithms

Keywords and phrases distributed graph algorithms, low-stretch spanning trees, CONGEST model, ball decomposition, star decomposition

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.4

Funding *Ruben Becker*: This work has been partially supported by the Italian MIUR PRIN 2017 Project ALGADIMAR "Algorithms, Games, and Digital Markets".
Yuval Emek: This work has been supported in part by an Israeli Science Foundation grant number 1016/17.

1 Introduction and Related Work

Trees are easy, general graphs are hard. This can be said as a first-order summary for a wide range of graph problems, especially in the area of approximation algorithms. Starting with the work of Alon et al. [3], there has been a beautiful line of developments that try to combat this issue and make general graphs (almost) as easy as trees, for several families of graph problems (including distances, cuts, and more) [5, 6, 8, 14, 7, 12, 29, 1, 13, 4, 26, 2]. In a very rough sense, these methods transform any general graph G to a tree T that approximately preserves some of the structural properties of G, thus opening the road for the following (generic) algorithmic approach: (1) transform the graph G into a tree T; (2) solve the problem on T; and (3) project the solution in T back to a solution in G. The quality of the obtained

© Ruben Becker, Yuval Emek, Mohsen Ghaffari, and Christoph Lenzen;
licensed under Creative Commons License CC-BY

solution depends on how well T preserves the relevant structure of G. Such transformations have been a key driver in many of the algorithmic developments in the past two decades, in centralized approximation algorithms. Our focus in this paper is on distance-related graph problems and transformations of graphs into *spanning trees* that approximately preserve distances, based on the notion of *stretch*.

Spanning Trees and Stretch

Consider some graph $G = (V, E, \ell)$, where $\ell : E \to \mathbb{R}_{>0}$ is an edge *length* function.[1] A tree $T = (V_T, E_T, \ell_T)$ is said to be a *spanning tree* of G if (i) $V_T = V$; (ii) $E_T \subseteq E$; and (iii) ℓ_T is the restriction of ℓ to the edges in E_T. By definition, the distances in T are at least as large as those in G, namely, $\mathrm{d}_T(u, v) \geq \mathrm{d}_G(u, v)$ for every two vertices $u, v \in V$, where the distances $\mathrm{d}_T(\cdot, \cdot)$ and $\mathrm{d}_G(\cdot, \cdot)$ are defined with respect to the edge length functions ℓ_T and ℓ, respectively. The notion of *stretch* provides a bound in the converse direction: given an edge $e = (u, v) \in G$, the *stretch* of e in T is defined to be $\mathrm{str}_T(e) = \mathrm{d}_T(u, v)/\ell(e)$.

Ideally, we would have wanted to construct a spanning tree T that admits a small stretch for every edge $e \in E$, but this is clearly hopeless, e.g., if the graph has high girth. Instead, we wish to construct a *random* spanning tree T so that the *expected stretch* of every edge $e \in E$ satisfies $\mathbb{E}_T[\mathrm{str}_T(e)] \leq \alpha$ for some small α (cf. [5]). This notion is closely related (and essentially equivalent) to constructing a (deterministic) spanning tree with *average stretch* α [3]. More precisely, the per-edge expected stretch guarantee trivially leads to a bound of $O(m \cdot \alpha)$ on the expected *total* stretch. Using standard techniques, this leads to a distributed construction of a spanning tree whose total stretch is $O(m \cdot \alpha)$ with high probability. Therefore, the per-edge expected stretch guarantee is sufficient for the method to be functional as a subroutine.

There is an extensive literature on constructing (random) spanning trees for general graphs with low expected stretch, starting with the pioneering work of Alon et al. [3] which paved the way for the developments in [3, 12, 1, 2]. The state-of-the-art in this line of work is Abraham and Neiman's construction of random spanning trees with expected stretch $O(\log n \log \log n)$ [2]. In a related line of work [5, 6, 8, 14, 7], it is only the distances in G that matter, essentially ignoring the graph topology so that the tree T can include vertices and edges that are not part of G, subject to the constraint that the distances in T are lower-bounded by the corresponding distances in G. The common practice here is to think of T as a *dominating tree metric* into which the metric space defined by the distances in G can be embedded without contracting the distances. The construction of Fakcharoenphol et al. [14] (often referred to as *FRT*) provides an asymptotically optimal $O(\log n)$ upper bound on the expected stretch in this setting (see also [7]).

Following the influential work of Bartal [5], random dominating tree metrics with low expected stretch have contributed greatly to the design of approximation and online algorithms, for problems in which the topology of the underlying graph G is abstracted away. More recently though there are new applications that require that T is a subgraph of G including fast solvers for symmetric diagonally dominant (SDD) linear systems [23, 21, 10, 11] and approximate max-flow and minimum cut algorithms [26, 9, 25, 31, 20], these applications point the flashlight back in the direction of low stretch spanning trees.

---

[1] Unless stated otherwise, all graphs in this paper are assumed to be undirected and finite.

Distributed Constructions

Low stretch spanning trees, as well as low stretch dominating tree metrics, have also been studied and used in distributed graph algorithms. For instance, low stretch spanning trees were a key component in the max-flow algorithm of Ghaffari et al. [18] which gave the first sublinear-time distributed max-flow approximation. However, currently known distributed constructions of these trees have a suboptimal running time and/or suboptimal stretch.

Khan et al. [22] were the first to investigate distributed algorithms for random dominating tree metrics with low expected stretch, designing a distributed implementation for the FRT construction that works in $\tilde{O}(SPD)$ rounds of the CONGEST model [28], where SPD denotes the shortest-path-diameter of the network, which can be as large as $\Theta(n)$, even in graphs with very small hop diameter D. Similar to many other graph problems, a lower bound of $\tilde{\Omega}(D + \sqrt{n})$ rounds follows from the work of Das Sarma et al. [30] which set $\tilde{O}(D + n^{0.5})$ as the target desired round complexity. Ghaffari and Lenzen [19] provided a faster distributed construction that runs in $\tilde{O}(D + n^{0.5+\varepsilon})$ rounds and builds a random dominating tree metric with expected stretch $O(\log n/\varepsilon)$. Friedrichs and Lenzen [16] further advanced this line of work by developing a distributed algorithm that outputs a random dominating tree metric with expected stretch $O(\log n)$ in $\tilde{O}((D + n^{0.5}) \cdot n^{o(1)})$ rounds.

The aforementioned distributed constructions suffer from two drawbacks: (1) the round complexity is still somewhat far from the $\tilde{\Omega}(D + \sqrt{n})$ target; and (2) the constructed trees do not have any guarantees regarding the topology of the underlying network and in particular, they are not spanning trees of this network, thus complicating their usage in distributed settings. For the more desirable, but also more stringent, notion of low stretch spanning trees, where the tree T has to be a subgraph of the G, the only known distributed construction is due to Ghaffari et al [18]. It gives a tree with a much worse stretch of $2^{O(\sqrt{\log n \cdot \log\log n})}$ and it runs in $\tilde{O}((D + n^{0.5}) \cdot 2^{\sqrt{\log n \cdot \log\log n}})$ rounds, both of which are a factor of $2^{O(\sqrt{\log n \cdot \log\log n})}$ away from ideal. Indeed, this suboptimality (in both stretch and running time) is one of the two bottlenecks in turning the round complexity of the max-flow approximation to the optimal bound of $\tilde{O}(D + \sqrt{n})$.

1.1 Our Contribution

We provide a new distributed algorithm, operating in the CONGEST model, that improves the state-of-the-art for low stretch spanning trees. For unweighted graphs (i.e., graphs with unit edge lengths), our algorithm runs in asymptotically optimal $O(D)$ rounds and builds a random spanning tree with expected stretch $O(\log^3 n)$. In terms of both round complexity and stretch, this improves considerably on the algorithm of Ghaffari et al [18] which has round complexity $\tilde{O}((D+n^{0.5}) \cdot 2^{\sqrt{\log n \cdot \log\log n}})$ and stretch $2^{O(\sqrt{\log n \cdot \log\log n})}$. Our algorithm also extends to weighted graphs with the same expected stretch guarantee, in which case the round complexity grows to $\tilde{O}(\min\{\sqrt{nD}, \sqrt{n}D^{1/4} + n^{3/5} + D\})$, i.e., boiling down to the best known complexity for exact single-source shortest path computation, which is due to Forster and Nanongkai [15]. We stress that this is the first distributed construction of spanning trees leading to poly-logarithmic expected stretch with non-trivial round complexity.

▶ **Theorem 1.** *A spanning tree of expected stretch $O(\log^3 n)$ for each edge can be computed in $\tilde{O}(\min\{\sqrt{nD}, \sqrt{n}D^{1/4} + n^{3/5} + D\})$ rounds w.h.p.*[2] *If the input graph is unweighted, then the same can be achieved in $O(D)$ rounds whp.*

[2] We say that event A occurs *with high probability*, abbreviated *w.h.p.*, if $\Pr(A) \geq 1 - n^{-c}$ for a constant c that can be made arbitrarily large.

More generally, our method can be seen as an efficient reduction of the task of computing a low-stretch spanning tree of expected stretch $O(\log^3 n)$ to single-source shortest path computations with a virtual super-source, which is formalized in Definition 2. Any improvements in distributed algorithms for this task will thus carry over to our construction.

We note that, unfortunately, our approach cannot be used to reduce to *approximate* single-source shortest path computations, as the decomposition technique that we will use [27] crucially relies on the subtractive form of the triangle inequality, which fails even under small relative errors in distances.

1.2 Our Method In a Nutshell

The general approach taken in the current paper is very similar to the divide and conquer technique due to Elkin et al. [12]. That is, we apply a graph partitioning scheme called *star decomposition* (introduced formally in Section 2) that given a root or center node x_0, decomposes the graph into a center part V_0 and *cone* parts $V_1, \dots, V_k$ centered at nodes $x_1, \dots, x_k$, respectively, referred to as the cone *anchors*. This star decomposition has the following properties: (1) the radius of V_i with respect to x_i, $0 \leq i \leq k$, is smaller than the radius r of G with respect to x_0 by a constant factor; and (2) each anchor node x_i, $i \in [k]$, is connected via a direct edge, referred to as a *bridge edge*, to some node $y_i \in V_0$ so that, for every cone, the distance between anchor node and center of the decomposition plus the radius of the cone is at most a factor of $1 + \varepsilon$ larger than the radius r with respect to x_0.

The idea is then to apply such star decompositions recursively to each of the obtained parts $V_0, \dots, V_k$, leading to spanning trees $T_0, \dots, T_k$. The spanning tree T that is returned by the algorithm is then constructed by connecting the trees $T_1, \dots, T_k$ to the central tree T_0 using the bridge edges $(x_1, y_1), \dots, (x_k, y_k)$. Clearly this approach leads to a spanning tree, however from the description so far, it is not clear why T has small expected stretch.

For this, we need that the star decompositions that we construct have the additional *small cut* property: For each edge $e = (u, v) \in E$, the probability that e is cut by the decomposition, i.e., that $u \in V_i$ and $v \in V_j$ for $i \neq j$, is at most $O(\log n \cdot \ell(e)/(\varepsilon r))$. Elkin et al. [12] use an intricate cone growing procedure in order to construct the parts $V_1, \dots, V_k$ in a way that ensures the (deterministic counterpart of the) small cut property. It is not clear though how to implement the cone growing procedure efficiently in a distributed manner.

In this paper, we replace the cone growing process of [12] by a graph partitioning technique due to Miller et al. [27]. This technique has the desirable property that it can be implemented in the CONGEST model of computation in a straightforward way based on single source shortest path (SSSP) computations. Specifically, after constructing the center part V_0, we let every node u that is "just outside" V_0 (we make this notion precise in Section 3) draw a value δ_u from an exponential distribution with mean $\beta = \Theta(\frac{\log n}{\varepsilon r})$. We now conceptually start a ball growing process from all such nodes, where node u joins the process at time δ_u.

Miller et al. [27] have shown (for the unweighted case) that this leads to a decomposition that "cuts edges" with a probability sufficiently small for their needs (they were not concerned with star decompositions). We observe that when applied to the graph $H = G \setminus V_0$, this leads to a star decomposition that satisfies the desired small cut property, see Section 3.2. In Section 4, we furthermore show that this is sufficient for the resulting tree (after recursing on the parts of the decomposition and connecting the obtained trees using the bridge edges) to have expected stretch $O(\log^3 n)$ for every edge. Replacing the cone growing process with this decomposition technique also results in a conceptually much simpler algorithm for constructing spanning trees of small expected stretch in standard centralized models of computation. This can be of independent interest. Lastly, we remark that, also for the PRAM model, our technique yields a similar reduction of computing spanning trees of low expected stretch to exact SSSP with virtual super-source.

2 Preliminaries

Our algorithm runs on an undirected, connected input graph $G = (V, E, \ell)$ with positive integer edge lengths ℓ that are polynomially bounded, i.e., bounded by n^c for some constant c. For graphs H, we denote by $\mathrm{d}_H(u, v)$ the length of the shortest path between two nodes u and v. If H is the input graph G, we may omit G and simply write $\mathrm{d}(u, v)$ for $\mathrm{d}_G(u, v)$.

For a node $u \in V$ and a radius $r > 0$, we call $B(u, r) := \{v \in V : \mathrm{d}(u, v) \leq r\}$ the ball of radius r around u, i.e., the set of nodes of distance at most r from u. For any graph G with node set V and a node $u \in V$, we call $\mathrm{rad}_u(G) = \max\{\mathrm{d}_G(u, v) : v \in V\}$ the radius of G with respect to u. For a subset of nodes $S \subset V$, we denote with $G[S]$ the subgraph of G induced by S. By $W(H) = \max_{u,v \in V_H}\{\mathrm{d}_H(u, v)\}$ we denote the weighted diameter of $H = (V_H, E_H, \ell_H)$ and by $D(H) = W(H')$ its hop diameter, where $H' = (V_H, E_H, \mathbf{1})$, i.e., H with all edges being assigned unit length; again, we omit G from the notation in case $H = G$.

Model of Computation

Our algorithm works in the standard CONGEST model of computation [28]. In this model, every node hosts a processor (of unlimited computational power) and is labeled by a unique $O(\log n)$-bit identifier. The computation proceeds in synchronous rounds, in each of which a node (1) performs local computations, (2) sends $O(\log n)$-bit messages to its neighbors, and (3) receives the messages that its neighbors sent. Initially, every node in the input graph $G = (V, E, \ell)$ knows its identifier and its incident edges together with their length. At the end of computation every node needs to know its part of the output. That is every node needs to know for its incident edges whether or not they belong to the output spanning tree.

Distributed SSSP Computation in Super-Source Graphs

At its heart, our algorithm reduces the problem to a series of single source shortest path (SSSP) computations. Accordingly, we will need to compute SSSP in graphs G_s that result from subgraphs of G by adding a (virtual) *super-source node* $s \notin V$.

▶ **Definition 2** (Super-source graphs). *Fix a subgraph $H = (V_H, E_H, \ell|_H)$ of G. Construct $G_s = (V_H \dot\cup \{s\}, E_H \cup E_s, \ell^{G_s})$ by choosing $E_s \subseteq V_H \times \{s\}$, picking $\ell^{G_s}(e) \in \{1, \ldots, n^c\}$ for $e \in E_s$, and setting $\ell^{G_s}(e) = \ell_e$ for all $e \in E_H$. We refer to G_s as a* super-source graph *(of G) and to s as its* super-source. *For distributed algorithms, we assume that each node $v \in V$ initially knows which of its incident edges in G are in V_H, whether it is connected to s, and, if so, the length of edge (s, v).*

Although both distributed CONGEST algorithms for SSSP that we employ (for the unweighted and weighted case) assume to be run on the input graph, we observe that it is straightforward to generalize them to super-source graphs. Both considered algorithms output a tree T that is a subgraph of G_s, where each node $v \in V_H$ learns its parent and the distance $\mathrm{d}_T(v, s)$ from v to s in T, which is exactly the distance $\mathrm{d}_{G_s}(v, s)$ from v to s in G_s.

▶ **Lemma 3** (folklore result). *SSSP in super-source graphs can be solved in $O(W(G_s))$ rounds. In particular, if G_s is unweighted (i.e., $\ell^{G_s} = \mathbf{1}$), SSSP can be solved in $O(D(G_s))$ rounds.*

Proof. For unweighted graphs, this is done by standard flooding to construct a BFS tree, where communication by s is simulated locally based on nodes knowing whether they are connected to s and by which length. For weighted graphs, one simulates the algorithm on the unweighted graph obtained by subdividing each edge e into ℓ_e many length-1 edges. Termination is detected via the resulting spanning forest $T \setminus \{s\}$ of $G_s \setminus \{s\}$. ◀

▶ **Corollary 4** (of [15]). *SSSP in super-source graphs can be solved in* $\tilde{O}(\min\{\sqrt{nD}, \sqrt{n}D^{1/4} + n^{3/5} + D\})$ *rounds w.h.p.*

▶ Comment. We comment that the algorithm of Forster and Nanongkai [15] can be directly extended to the case of super-source graphs.[3] In short, the reason is as follows: there are only two differences between the case considered here and the one of [15]. (1) We cannot communicate on the virtual edges that connect s to V_H, as there are no such physical edges. (2) We work on a subgraph of the base graph, whose hop diameter may be much larger than D. Regarding the first point, we note that in the algorithm of [15], besides the initial coordination message from the source that can be delivered to all nodes in $O(D)$ rounds, the source s never changes its state. Hence, it does not need to send any message to its neighbors in V_H or to receive a message from them. Regarding the second point, we note that the algorithm of [15] relies on the hop diameter D only for the purpose of global communication. In our setting, even though our computation is about a subgraph, we can still use the base graph to perform computation and in particular we can deliver any B messages to all nodes in $O(D + B)$ rounds.

Exponential Distribution

By Exp_β we denote the exponential distribution with mean $1/\beta$. Its density function is given by $f_{\mathrm{Exp}_\beta}(x) = \beta \exp(-\beta x) \cdot H(x)$, where $H(\cdot)$ denotes the Heaviside step function and its cumulative density function by $F_{\mathrm{Exp}_\beta}(x) = (1 - \exp(-\beta x)) \cdot H(x)$.[4] First, we observe that drawing from this distribution results in values of $O(\log n/\beta)$ w.h.p.

▶ **Lemma 5.** *For parameters* $0 < \varepsilon < 1$, $\beta > 0$, *and a sufficiently large constant* $c > 0$, *let* $t := c \log n/(4(1 + \varepsilon)\beta)$ *and* $X \sim \mathrm{Exp}_\beta$. *Then* $P[X \geq t] \in n^{-\Omega(c)}$, *i.e.,* $X < t$ *w.h.p.*

Proof. The proof is a simple calculation:

$$P[X \geq t] = \frac{\int_t^\infty e^{-\beta x}\,dx}{\int_0^\infty e^{-\beta x}\,dx} = \frac{e^{-\beta t}\int_0^\infty e^{-\beta x}\,dx}{\int_0^\infty e^{-\beta x}\,dx} \in e^{-\Omega(c \log n)} = n^{-\Omega(c)}\,. \qquad ◀$$

Intuitively, the next lemma (taken from [27]) is used as follows. Imagine that ball centers $u \in S \subseteq V$ each grow a ball independently and in parallel, but with starting times shifted by $-\delta_u$. Then, no matter how far exactly the ball centers are from a given edge e in the graph, the arrival times of the first and second ball differ by at least $2\ell_e$ with probability $1 - O(\beta\ell_e) = 1 - O(\frac{\ell_e \log n}{\varepsilon r})$, using $\beta = \Theta(\frac{\log n}{\varepsilon r})$. For any edge e of length ℓ_e, this means that the ball arriving first at one endpoint of the edge also is the first to arrive at the other endpoint with probability at least $1 - O(\frac{\ell_e \log n}{\varepsilon r})$.

▶ **Lemma 6** (Lemma 4.4 in [27]). *Let* $d_1 \leq \ldots \leq d_s$ *be arbitrary values and* $\delta_1, \ldots, \delta_s$ *be independent random variables picked from* Exp_β. *Then the probability that the smallest and the second smallest values of* $d_i - \delta_i$ *are within* c *of each other is at most* $O(\beta c)$.

[3] Verified through personal communication with Sebastian Forster and Danupon Nanongkai.
[4] Here the Heaviside step function is defined as $H(x) = 0$ if $x < 0$ and $H(x) = 1$ otherwise.

3 Computing a $(1/3, \varepsilon)$-Star Decomposition

We formally introduce (δ, ε)-star decompositions following their presentation in [12].

▶ **Definition 7.** *We call a partition $V_0, \ldots, V_k$ of V that satisfies*

(a) *for all $i \in [k] \cup \{0\} : G[V_i]$ is connected,*

(b) *for all $i \in [k]$ there is $e_i = (x_i, y_i) \in E$ with $x_i \in V_i$, $y_i \in V_0$*

a (δ, ε)-star decomposition, if, in addition,

(1) *$r_0 \leq (1 - \delta)r$, using the notation $r_i = \mathrm{rad}_{x_i}(G[V_i])$ and $r = \mathrm{rad}_{x_0}(G)$*

(2) *for all $i \in [k] : \mathrm{d}(x_0, x_i) \geq \delta r$*

(3) *for all $i \in [k] : \mathrm{d}(x_0, x_i) + r_i \leq (1 + \varepsilon)r$.*

We call the nodes $x_1, \ldots, x_k$ the *anchor nodes* of the parts $V_1, \ldots, V_k$ and the edges $e_1, \ldots, e_k$ are called the *bridge edges*. We refer the reader to Figure 1 for an illustration. Note also that properties (2) and (3) imply that the radius of each of the graphs $G[V_1], \ldots, G[V_k]$ is upper bounded by $(1 + \varepsilon - \delta) \cdot r$.

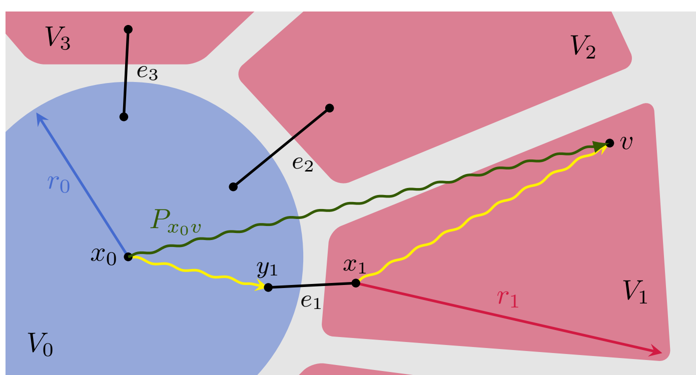


Figure 1 An illustration of a (δ, ε)-star decomposition. The center part V_0 of radius $r_0 \leq (1 - \delta)r$ is connected to each part V_i via a bridge edge $e_i = (x_i, y_i)$. The anchor nodes x_i have distance at least δr to x_0. Moreover, the distance of x_0 to x_i with $i \geq 1$ plus the radius r_i of the part V_i is at most $(1 + \varepsilon)$ times the radius r of the original graph. Note that the decomposition that we construct leads the stronger property that, for any node $v \in V_i$, the length of the path from x_0 to v over the bridge edge e_i (drawn in yellow) is at most εr longer than the shortest path $P_{x_0 v}$ from x_0 to v (drawn in green).

Given a root node $x_0 \in V$ and a radius r_0, let $V_0 := B(x_0, r_0)$. We let $S := \{u \in V \setminus V_0 : \exists v \in V_0, (u, v) \in E \text{ and } \mathrm{d}(x_0, u) = \mathrm{d}(x_0, v) + \ell_{(u,v)}\}$ be the so-called *ball-shell of* V_0, i.e., the nodes u outside V_0 that have a neighbor v in V_0 such that a shortest path from x_0 to u passes through v. For a node $u \in S$, we fix v_0^u to be some neighbor of u in V_0 such that $\mathrm{d}(x_0, u) = \mathrm{d}(x_0, v_0^u) + \ell_{(v_0^u, u)}$.

Now, let $\delta : S \to \mathbb{R}_{\geq 0}$ be a function that assigns a non-negative real to every node on the ball-shell. For every node $u \in S$, we define the *adjusted δ-shifted distance* of u as

$$\mathrm{ad}_{x_0}^{-\delta}(u) := \mathrm{d}(x_0, u) + \max_{v \in S}\{\delta_v\} - \delta_u \,.$$

We remark that the shift by $\max_{v \in S}\{\delta_v\}$ is simply used in order to ensure non-negativity. These numbers define the delay after which u starts to grow its ball (if it has not yet joined another ball), which we adjust compared to [27] by the distance of u to x_0. This adjustment allows us to treat the general weighted case; in the unweighted setting, all of these distances would be identical as u is a node on the ball-shell of V_0 and thus the resulting decomposition would remain unaffected by the adjustment.

Algorithm 1 `star_decompose`(G, x_0, ε).

Input : graph $G = (V, E, \ell)$, node $x_0 \in V$, $\varepsilon > 0$
Output : $(1/3, \varepsilon)$-star decomposition of G w.h.p.

1 Compute $r = \text{rad}_{x_0}(G)$.
2 Set $\beta := \frac{c \log n}{\varepsilon r}$, sample $r_0 \in [\frac{r}{2}, \frac{2r}{3}]$ u.a.r. // c is a suff. large constant
3 Let $V_0 = B(x_0, r_0)$ and $H := G[V \setminus V_0]$.
4 Let $S := \{u \in V \setminus V_0 : \exists v \in V_0, (u, v) \in E \text{ and } \text{d}(x_0, u) = \text{d}(x_0, v) + \ell_{(u,v)}\}$.
5 For each $u \in S$, pick $\delta_u \sim \text{Exp}_\beta$ independently.
6 $G_s :=$ super-source graph obtained from H by attaching $u \in S$ to s with length $\text{ad}_{x_0}^{-\delta}(u)$.
7 Compute SSSP tree T of G_s rooted at s.
8 Let $x_1, \ldots, x_k$ be the children of s in T and $V_1, \ldots, V_k$ be the node sets of their subtrees.
9 For each x_i let $y_i = v_0^{x_i} \in V_0$
10 **return** sets $V_0, V_1, \ldots, V_k$, anchors $x_1, \ldots, x_k$, nodes $y_1, \ldots, y_k$

We now describe Algorithm 1, which computes a $(1/3, \varepsilon)$-star decomposition. The algorithm starts by carving out the center ball around x_0, which has a randomized radius r_0 to ensure that edges e are cut with probability $O(\ell_e/r)$. It then grows balls in $G[V \setminus V_0]$ around the shell nodes S, where the starting times are delayed according to random shifts $\delta_v \sim \text{Exp}_\beta$ (this is the technique from [27]). Here, $\beta \in \tilde{\Theta}(1/r)$ with r being the radius of G w.r.t. x_0, so that the probability to cut an edge e of length ℓ_e is $\tilde{O}(\ell_e/r)$. This is implemented by an SSSP computation with super-source s, which is attached to shell node u by an edge of length $\text{ad}_{x_0}^{-\delta}(u)$, which results in the desired behavior. The subtrees rooted at children of s then correspond to the balls, and the algorithm can return the desired decomposition.

For ease of presentation, we assume in our analysis the non-integrality of the δ_u's is not an issue for the single source shortest path computations used; it is straightforward to use values that are rounded to integers.[5]

▶ **Corollary 8.** *Algorithm 1 can be implemented in* $\tilde{O}(\min\{\sqrt{nD}, \sqrt{n}D^{1/4} + n^{3/5} + D\})$ *rounds w.h.p. If* G *is unweighted, it can be implemented in* $O(D)$ *rounds w.h.p.*

Proof. From the pseudo-code of the algorithm, it is immediate that all computations are local except (i) determining $\text{rad}_{x_0}(G)$, (ii) finding $B(x_0, r_0)$, (iii) determining T, and (iv) determining the subtrees of $T \setminus \{s\}$. (i) and (ii) can be performed by a call to an SSSP algorithm (a single call suffices, in fact) and making r_0 known to all nodes. The same holds true for (iii). Regarding (iv), in order to determine the connected components of $T \setminus \{s\}$, we can invoke a variant of the minimum spanning tree algorithm by Garay, Kutten and Peleg [17, 24], which runs in $\tilde{O}(\sqrt{n} + D)$. Applying Corollary 4, the first claim follows.

[5] This can be interpreted as distorting edge lengths by $O(1)$ in our analysis (possibly even inconsistenly in different bounds). As all probability bounds involving edge lengths ℓ_e are asymptotic and linear in ℓ_e and the minimum edge length is 1, there is no change in the asymptotic results.

If G is unweighted, the first SSSP computation has running time $O(D)$ by Lemma 3. The same holds for the second, provided that $W(G_s) \in O(D)$. By Lemma 5, $\max_{u \in S}\{\delta_u\} \in O(\log n/\beta) \subset O(D)$ w.h.p. Thus,

$$W(G_s) \leq 2\,\mathrm{rad}_s(G_s) \leq 2(\max_{u \in S}\{\delta_u\} + \mathrm{rad}_{x_0}(G)) \in O(D)$$

w.h.p. As the depth of T is at most $W(G_s)$, the naive algorithm for finding the connectivity components of $T \setminus \{s\}$ completes in $O(D)$ rounds w.h.p. as well. ◀

3.1 Correctness

We now show that Algorithm 1 indeed returns a $(1/3, \varepsilon)$-star decomposition of G w.h.p.

▶ **Theorem 9.** *Algorithm 1 outputs a $(1/3, \varepsilon)$-star decomposition w.h.p.*

In order to prove the theorem, we examine the conditions in Definition 7. First, observe that the graphs $G[V_i]$ are spanned by the components of $T \setminus \{s\} \subset V$ and thus connected (in G). Therefore, condition (a) of the (δ, ε)-star decomposition holds by construction. In particular, $r_i := \mathrm{rad}_{x_i}(G[V_i])$ is well-defined for every $i \in [k]$. Similarly, (b) is immediate from the construction, (1) holds since $r_0 \leq 2r/3$, and (2) holds since $r_0 \geq r/2 > r/3$. We show that condition (3) holds w.h.p., the proof is based on Lemma 5.

▶ **Lemma 10.** *Let $x_0 \in V$ be the root node given to the algorithm. Moreover, let $V_0, V_1, \ldots, V_k$, S, and $x_0, x_1, \ldots, x_k$, as well as $y_1, \ldots, y_k$ be as in Algorithm 1. Let $r = \mathrm{rad}_{x_0}(G)$ and G' be the subgraph of G in which all edges in $V_i \times V_j$ for $i \neq j \in \{0, \ldots, k\}$ are deleted except for the bridge edges $(x_1, y_1), \ldots, (x_k, y_k)$. Then $\mathrm{rad}_{x_0}(G') \leq (1 + \varepsilon) \cdot r$ w.h.p., i.e., property (3) of Definition 7 holds.*

Proof. For arbitrary $v \in V \setminus V_0$, denote by $x_i \in S$ a shell node such that $v \in V_i$. As $H = G[V \setminus V_0]$ is a subgraph of G_s and $\mathrm{d}(x_i, v) = \mathrm{d}_H(x_i, v)$ by construction, we have that

$$\mathrm{d}_{G'}(x_0, v) = \mathrm{d}(x_0, y_i) + \ell_{(x_i, y_i)} + \mathrm{d}(x_i, v) = \mathrm{d}(x_0, x_i) + \mathrm{d}_H(x_i, v) \leq \mathrm{d}_{G_s}(x_0, v)\,.$$

By Lemma 5 and a union bound, $\max_{w \in S}\{\delta_w\} < c \log n/\beta \leq \varepsilon r$ w.h.p. Let $u \in S$ be a shell-node on a shortest path from x_0 to v, i.e., $\mathrm{d}(x_0, v) = \mathrm{d}(x_0, u) + \mathrm{d}(u, v)$. Then, w.h.p.,

$$\begin{aligned}\mathrm{d}_{G_s}(x_0, v) \leq \mathrm{d}_{G_s}(x_0, u) + \mathrm{d}_{G_s}(u, v) &\leq \mathrm{d}(x_0, u) + \max_{w \in S}\{\delta_w\} - \delta_u + \mathrm{d}_H(u, v) \\ &< \mathrm{d}(x_0, u) + \varepsilon r + \mathrm{d}(u, v) = \mathrm{d}(x_0, v) + \varepsilon r \leq (1 + \varepsilon) r\,,\end{aligned}$$

where we again used that $H = G[V \setminus V_0]$ is a subgraph of G_s and $\mathrm{d}(u, v) = \mathrm{d}_H(u, v)$ by construction. As G' preserves distances to all nodes in $V_0 \cup S$, the claimed bound on the radius follows. Note that property (3) of Definition 7 follows as well, as in G' the edges $(x_1, y_1), \ldots, (x_k, y_k)$ are bridges and thus

$$\mathrm{rad}_{x_0}(G') = \max_{i \in [k]}\{\mathrm{d}(x_0, x_i) + \mathrm{rad}_{x_i}(G[V_i])\}\,.$$ ◀

3.2 Probability To Cut an Edge

Given a (δ, ε)-star decomposition $V_0, \ldots, V_k$, we say that an edge is cut if its endpoints belong to different parts V_i of the decomposition. There are two different ways in which an edge can be cut by Algorithm 1. (1) It can be cut by the process of growing the center V_0 or (2) it is cut during the procedure in lines 5-8. We call $E_\circ^{\mathrm{cut}} := \{(u, v) \in E : u \in V_0, v \in V_i, i \in [k]\}$,

i.e. edges between V_0, V_i for $i \in [k]$ the edges resulting from (1) and $E^{\text{cut}}_{\triangle} := \{(u,v) \in E : u \in V_i, v \in V_j, i, \neq j \in [k]\}$, i.e. edges between V_i, V_j for $i, j \in [k]$ the cut edges resulting from (2). Moreover we let $E^{\text{cut}} = E^{\text{cut}}_{\circ} \cup E^{\text{cut}}_{\triangle}$. The goal of this subsection is to prove the following lemma. Its proof is split in two lemmata, Lemma 12 and Lemma 13.

▶ **Lemma 11.** *For an edge $e = (u,v) \in E$, it holds that $\Pr[e \in E^{\text{cut}}] = O(\frac{\ell_e \log n}{\varepsilon r})$.*

It is simple to bound the probability of an edge e being in $E^{\text{cut}}_{\circ}$:

▶ **Lemma 12.** *For an edge $e = (u,v) \in E$, it holds that $\Pr[e \in E^{\text{cut}}_{\circ}] \leq \frac{6\ell_e}{r} = O(\frac{\ell_e}{r})$.*

Proof. W.l.o.g. assume that $\mathrm{d}(x_0,u) \leq \mathrm{d}(x_0,v)$. It then follows that

$$\Pr[e \in E^{\text{cut}}_{\circ}] \leq \Pr[r_0 \in [\mathrm{d}(x_0,u), \mathrm{d}(x_0,u) + \ell_e)] = \frac{6\ell_e}{r} = O\Big(\frac{\ell_e}{r}\Big),$$

since r_0 was picked u.a.r. from the interval $[r/2, 2r/3]$ of width $r/6$. ◀

We now wish to bound the probability that an edge $e = (u,v) \in E$ belongs to $E^{\text{cut}}_{\triangle}$. Essentially, the desired bound is implicit in [27], but due to our modifications for the weighted setting we cannot apply the statements from this work as a black box entirely. However, the statement follows without much effort from Lemma 6.

▶ **Lemma 13.** *For an edge $e = (u,v) \in E$, it holds that $\Pr[e \in E^{\text{cut}}_{\triangle}] \in O(\frac{\ell_e \log n}{\varepsilon r})$.*

Proof. Recall that $H = G[V \setminus V_0]$. If $(u,v) \notin E_H$, then $(u,v) \notin E^{\text{cut}}_{\triangle}$, so assume that $e = (u,v) \in E_H$. Denote by $x_u \in S$ the anchor node of the part u belongs to, i.e., $\mathrm{d}_{G_s}(x_0,u) = \mathrm{d}(x_0,x_u) + \max_{x\in S}\{\delta_x\} - \delta_{x_u} + \mathrm{d}(x_u,u)$. We apply Lemma 6 to the values $d_x = \mathrm{d}(x_0,x) + \mathrm{d}(x,u)$ and δ_x chosen in line 5, where $x \in S$. This shows that with probability $1 - O(\beta\ell_e)$, we have for all $x \neq x_u$ that

$$\mathrm{d}(x_0,x) + \mathrm{d}(x,u) - \delta_x > \mathrm{d}(x_0,x_u) + \mathrm{d}(x_u,u) - \delta_{x_u} + 2\ell_e\,.$$

Adding $\max_{x\in S}\{\delta_x\} - \ell_e$ on both sides of this inequality and applying the triangle inequality, this entails that

$$\begin{aligned}\mathrm{ad}^{-\delta}_{x_0}(x_u) + \mathrm{d}(x_u,v) &\leq \mathrm{d}(x_0,x_u) - \delta_{x_u} + \max_{x\in S}\{\delta_x\} + \mathrm{d}(x_u,u) + \ell_e \\ &< \mathrm{d}(x_0,x) + \mathrm{d}(x,u) - \delta_x - \ell_e + \max_{x\in S}\{\delta_x\} \leq \mathrm{ad}^{-\delta}_{x_0}(x) + \mathrm{d}(x,v)\end{aligned}$$

for all $x \in S \setminus \{x_u\}$. It follows that the shortest path from x_0 to v in G_s passes through x_u, i.e., v is in the same part as u and e is not cut. Accordingly, the probability that $e \in E^{\text{cut}}_{\triangle}$ is bounded by $O(\beta\ell_e) = O(\frac{\ell_e \log n}{\varepsilon r})$, as claimed. ◀

4 Building the Low-Stretch Spanning Tree

We now give an algorithm that uses our $(1/3,\varepsilon)$-star decomposition algorithm recursively in order to compute a spanning tree of low expected stretch, see Algorithm 2. As described above, the algorithm takes as input the graph $G = (V,E,\ell)$ with n nodes and a root node $x_0 \in V$ (x_0 can be chosen arbitrarily) and outputs a tree T such that $\mathrm{E}[\mathrm{str}_T(e)] = O(\log^3 n)$ for any edge $e \in E$.

Consider an invocation of Algorithm 2 on an input graph G with root node $x_0 \in V$. Let $(H^{(0)}, x^{(0)}), (H^{(1)}, x^{(1)}), \dots$ with $G = H^{(0)}$ and $x_0 = x^{(0)}$ be a sequence of graphs and root nodes corresponding to a path in the recursion tree. Moreover, let $r^{(k)} = \mathrm{rad}_{x^{(k)}}(H^{(k)})$ be the radius of the k'th graph in this sequence with respect to $x^{(k)}$.

Algorithm 2 `low_stretch_tree`(G, x_0).

Input : graph $G = (V, E, \ell)$ with n nodes, root node $x_0 \in V$
Output : spanning tree T s.t. $\mathrm{E}[\mathrm{str}_T(e)] = O(\log^3 n)$ for every edge $e \in E$

```
1 ε = min{1/12, 1/log n}
2 if |V| ≤ 2 then
3     return G
4 else
5     (V_0, ..., V_k, x_1, ..., x_k, y_1, ..., y_k) = star_decompose(G, x_0, ε)
6     for i ∈ [k] ∪ {0} do
7         T_i = low_stretch_tree(G[V_i], x_i)
8     return T = ⋃_{i∈[k]∪{0}} T_i ∪ ⋃_{i∈[k]} {(x_i, y_i)}
```

▶ **Corollary 14.** *The recursion depth of Algorithm 2 is $O(\log n)$ w.h.p.*

Proof. By Theorem 9, each call to Algorithm 1 returns a $(1/3, \varepsilon)$-star decomposition w.h.p. Condition on these events. Denoting $r = \mathrm{rad}_{x_0}(G)$, from properties (1) and (2) of a $(1/3, \varepsilon)$-star decomposition we get by induction that $(\frac{2}{3})^i r = \max\{1-\delta, \delta\}^i r$ is a bound on the radius of subgraphs the algorithm is called on in recursion depth i. As edge lenghts are polynomially bounded, we have that $r \in n^{O(1)}$, implying that there is $i_{\max} \in O(\log n)$ so that $(\frac{2}{3})^i r < 1$. As the minimum edge length is 1, this entails that such a graph contains no edge and the recursion stops. As the number of recursive calls in depth i is trivially bounded by n (the maximum number of disjoint, non-empty subgraphs of G), we conditioned on $O(n \log n)$ events that occur w.h.p. By a union bound, the claim follows. ◀

We will now shift focus towards the following sequence of recursively defined graphs

$$R^{(0)}(G) := G, \quad \text{and} \quad R^{(t)}(G) := \bigcup_{i\in[k]\cup\{0\}} R^{(t-1)}(G)[V_i] \cup \bigcup_{i\in[k]} \{(x_i, y_i)\} \quad \text{for } t \geq 1,$$

for some graph G and partition $V_0, \ldots, V_k$ of its node set. Note that the defined sequence of $R^{(i)}(G)$ becomes sparser and sparser until the final one is the tree returned by the algorithm. As opposed to the sequence $H^{(0)}, H^{(1)}, \ldots$ that we considered previously (corresponding to a recursive path in the recursion tree) however, each of the graphs in $R^{(0)}(G), R^{(1)}(G), \ldots$ contains all the nodes of G. In fact, $R^{(\ell)}(G)$ is the graph that we would obtain when interrupting Algorithm 2 at recursion level ℓ.

▶ **Lemma 15.** *For a graph G and the sequence $R^{(i)}(G)$ as defined above, let $\rho^{(k)} := \mathrm{rad}_{x_0}(R^{(k)}(G))$ and let $\gamma^{(k)} = \rho^{(k)}/\rho^{(k-1)}$. Then*

$$\mathrm{E}[\gamma^{(k)}] \leq 1 + 2\varepsilon \quad \text{and} \quad \mathrm{E}[\rho^{(k)}] \leq (1+2\varepsilon)^k \cdot \mathrm{rad}_{x_0}(G)\,.$$

Proof. By Lemma 10, we have $\Pr[\rho^{(k)} \leq (1+\varepsilon)\rho^{(k-1)}] = \Pr[\gamma^{(k)} \leq 1+\varepsilon] \geq 1 - n^{-c}$ for a constant c under our control. Choosing c sufficiently large, thus

$$\mathrm{E}[\gamma^{(k)}] \leq (1+\varepsilon) + n \cdot n^{-c} \leq 1 + 2\varepsilon, \quad \text{using } \varepsilon = 1/\log n \geq n^{-c+1}\,.$$

For the second claim, we get

$$\mathrm{E}[\rho^{(k)}] = \mathrm{E}\left[\rho^{(0)} \cdot \prod_{i=1}^{k} \gamma^{(k)}\right] = \rho^{(0)} \cdot \prod_{i=1}^{k} \mathrm{E}[\gamma^{(k)}] \leq (1+2\varepsilon)^k \cdot \mathrm{rad}_{x_0}(G)\,,$$

since the events are independent. ◀

▶ **Lemma 16.** $\mathrm{E}[\mathrm{str}_T(e)] = O(\log^3 n)$ *for the expected stretch of each edge* $e = (u, v) \in E$.

Proof. By Corollary 14, the recursion depth λ of Algorithm 2 satisfies $\lambda \in O(\log n)$ w.h.p. W.l.o.g., condition on this event.[6] Let us denote with E_k the set of edges being cut at recursive level k and let $d := \mathrm{d}_T(u, v)$ for notational convenience. Note that the event that edge e is cut in depth i of the recursion is disjoint from the event that it is cut in depth $j \neq i$. Hence, using the law of total expectation and Lemma 11, we get that

$$\mathrm{E}[d] = \mathrm{E}[d \,|\, e \in T] + \sum_{k=0}^{\lambda} \mathrm{E}[d \,|\, e \in E_k] \cdot \Pr[e \in E_k] = \ell_e + \sum_{k=0}^{\lambda} \mathrm{E}[d \,|\, e \in E_k] \cdot \Pr[e \in E_k] \,.$$

Consider the special case that $e \in E_0$. Clearly, $\mathrm{E}[d \,|\, e \in E_0] \leq \mathrm{E}[2\,\mathrm{rad}_{x_0} R^{\lambda}(G)]$ (i.e., twice the radius of the computed spanning tree) in the notation of Lemma 15. By the lemma and the fact that $\varepsilon \leq 1/\log n$, we get that

$$\mathrm{E}[d \,|\, e \in E_0] \leq \mathrm{E}[2\,\mathrm{rad}_{x_0}(R^{\lambda}(G))] \leq 2(1+2\varepsilon)^{\lambda} \cdot \mathrm{rad}_{x_0}(G) \in O(\mathrm{rad}_{x_0}(G)) \,.$$

Applying Lemma 11, we arrive at $\mathrm{E}[d \,|\, e \in E_0] \cdot \Pr[e \in E_0] \in O\left(\frac{\ell_e \log n}{\varepsilon}\right) = O(\ell_e \log^2 n)$. Now consider the case that $e \in E_k$ for some $k \neq 0$. Thus, e was contained in some connected subgraph H of G that Algorithm 2 was called on recursively. The same reasoning hence shows that $\mathrm{E}[d \,|\, e \in E_k] \cdot \Pr[e \in E_k] \in O(\ell_e \log^2 n)$. We conclude that

$$\mathrm{E}[\mathrm{str}_T(e)] = \frac{E[d]}{\ell_e} = 1 + \frac{\sum_{k=0}^{\lambda} \mathrm{E}[d \,|\, e \in E_k] \cdot \Pr[e \in E_k]}{\ell_e} \in 1 + O(\lambda \log^2 n) \subseteq O(\log^3 n) \,.$$ ◀

We now have everything in place to infer Theorem 1.

Proof of Theorem 1. By Lemma 16, Algorithm 2 achieves the stated guarantee on the stretch. By Corollary 14, its recursion depth is $O(\log n)$ w.h.p. Each call performs a call to Algorithm 1, which can be implemented with the stated running time bound by Corollary 8. Observe that we can perform for each SSSP computation step of Algorithm 1 on recursion level i the computation for all instances on this recursion level with a *single* instance of the SSSP algorithm we use: we perform the computation on the union of the subgraphs induced by disjoint sets, yielding for each subgraph the correct tree T by deleting all nodes not belonging to the induced subgraph. By Corollary 8, the stated running time bounds follow, where in the unweighted case we exploit that radii (and thus diameters) decrease exponentially with the recursion depth (cf. Corollary 14). ◀

References

1. Ittai Abraham, Yair Bartal, and Ofer Neiman. Nearly Tight Low Stretch Spanning Trees. In *49th Annual IEEE Symposium on Foundations of Computer Science, FOCS*, pages 781–790, 2008.
2. Ittai Abraham and Ofer Neiman. Using petal-decompositions to build a low stretch spanning tree. In *Proceedings of the 44th Symposium on Theory of Computing Conference, STOC*, pages 395–406, 2012.
3. Noga Alon, Richard M. Karp, David Peleg, and Douglas B. West. A Graph-Theoretic Game and Its Application to the k-Server Problem. *SIAM J. Comput.*, 24(1):78–100, 1995.

[6] As edge weights are from $1, \ldots, n^{O(1)}$, stretch is trivially bounded by $n^{O(1)}$. By choosing the constant c large enough, the expected contribution of events that occur with probability at most $1/n^c$ can be made negligible.

4 Reid Andersen and Uriel Feige. Interchanging distance and capacity in probabilistic mappings. *CoRR*, abs/0907.3631, 2009. `arXiv:0907.3631`.

5 Yair Bartal. Probabilistic Approximations of Metric Spaces and Its Algorithmic Applications. In *37th Annual Symposium on Foundations of Computer Science, FOCS*, pages 184–193, 1996.

6 Yair Bartal. On Approximating Arbitrary Metrices by Tree Metrics. In *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing, STOC*, pages 161–168, 1998.

7 Yair Bartal. Graph Decomposition Lemmas and Their Role in Metric Embedding Methods. In *Algorithms - ESA 2004, 12th Annual European Symposium*, pages 89–97, 2004.

8 Moses Charikar, Chandra Chekuri, Ashish Goel, Sudipto Guha, and Serge A. Plotkin. Approximating a Finite Metric by a Small Number of Tree Metrics. In *39th Annual Symposium on Foundations of Computer Science, FOCS*, pages 379–388, 1998.

9 Paul Christiano, Jonathan A. Kelner, Aleksander Madry, Daniel A. Spielman, and Shang-Hua Teng. Electrical flows, laplacian systems, and faster approximation of maximum flow in undirected graphs. In *Proceedings of the 43rd ACM Symposium on Theory of Computing, STOC*, pages 273–282, 2011.

10 Michael B. Cohen, Brittany Terese Fasy, Gary L. Miller, Amir Nayyeri, Richard Peng, and Noel Walkington. Solving 1-Laplacians in Nearly Linear Time: Collapsing and Expanding a Topological Ball. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 204–216, 2014.

11 Michael B. Cohen, Rasmus Kyng, Gary L. Miller, Jakub W. Pachocki, Richard Peng, Anup B. Rao, and Shen Chen Xu. Solving SDD linear systems in nearly $m\log^{1/2}n$ time. In *Symposium on Theory of Computing, STOC*, pages 343–352, 2014.

12 Michael Elkin, Yuval Emek, Daniel A. Spielman, and Shang-Hua Teng. Lower-Stretch Spanning Trees. *SIAM J. Comput.*, 38(2):608–628, 2008.

13 Matthias Englert and Harald Räcke. Oblivious Routing for the Lp-norm. In *50th Annual IEEE Symposium on Foundations of Computer Science, FOCS*, pages 32–40, 2009.

14 Jittat Fakcharoenphol, Satish Rao, and Kunal Talwar. A tight bound on approximating arbitrary metrics by tree metrics. *J. Comput. Syst. Sci.*, 69(3):485–497, 2004.

15 Sebastian Forster and Danupon Nanongkai. A Faster Distributed Single-Source Shortest Paths Algorithm. In *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS*, pages 686–697, 2018.

16 Stephan Friedrichs and Christoph Lenzen. Parallel Metric Tree Embedding Based on an Algebraic View on Moore-Bellman-Ford. *J. ACM*, 65(6):43:1–43:55, 2018.

17 Juan A Garay, Shay Kutten, and David Peleg. A sublinear time distributed algorithm for minimum-weight spanning trees. *SIAM Journal on Computing*, 27(1):302–316, 1998.

18 Mohsen Ghaffari, Andreas Karrenbauer, Fabian Kuhn, Christoph Lenzen, and Boaz Patt-Shamir. Near-Optimal Distributed Maximum Flow: Extended Abstract. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC*, pages 81–90, 2015.

19 Mohsen Ghaffari and Christoph Lenzen. Near-Optimal Distributed Tree Embedding. In *Distributed Computing - 28th International Symposium, DISC*, pages 197–211, 2014.

20 Jonathan A. Kelner, Yin Tat Lee, Lorenzo Orecchia, and Aaron Sidford. An Almost-Linear-Time Algorithm for Approximate Max Flow in Undirected Graphs, and its Multicommodity Generalizations. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 217–226, 2014.

21 Jonathan A. Kelner, Lorenzo Orecchia, Aaron Sidford, and Zeyuan Allen Zhu. A simple, combinatorial algorithm for solving SDD systems in nearly-linear time. In *Symposium on Theory of Computing Conference, STOC*, pages 911–920, 2013.

22 Maleq Khan, Fabian Kuhn, Dahlia Malkhi, Gopal Pandurangan, and Kunal Talwar. Efficient distributed approximation algorithms via probabilistic tree embeddings. *Distributed Computing*, 25(3):189–205, 2012.

23 Ioannis Koutis, Gary L. Miller, and Richard Peng. A Nearly-m log n Time Solver for SDD Linear Systems. In *IEEE 52nd Annual Symposium on Foundations of Computer Science, FOCS 2011, Palm Springs, CA, USA, October 22-25, 2011*, pages 590–598, 2011.

24 Shay Kutten and David Peleg. Fast Distributed Construction of Smallk-Dominating Sets and Applications. *Journal of Algorithms*, 28(1):40–66, 1998.

25 Yin Tat Lee, Satish Rao, and Nikhil Srivastava. A new approach to computing maximum flows using electrical flows. In *Symposium on Theory of Computing Conference, STOC*, pages 755–764, 2013.

26 Aleksander Madry. Fast Approximation Algorithms for Cut-Based Problems in Undirected Graphs. In *51th Annual IEEE Symposium on Foundations of Computer Science, FOCS*, pages 245–254, 2010.

27 Gary L. Miller, Richard Peng, and Shen Chen Xu. Parallel graph decompositions using random shifts. In *25th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '13, Montreal, QC, Canada - July 23 - 25, 2013*, pages 196–203, 2013.

28 D. Peleg. *Distributed Computing: A Locality-Sensitive Approach*. Discrete Mathematics and Applications. Society for Industrial and Applied Mathematics, 2000.

29 Harald Räcke. Optimal hierarchical decompositions for congestion minimization in networks. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing, STOC*, pages 255–264, 2008.

30 Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. Distributed Verification and Hardness of Distributed Approximation. *SIAM J. Comput.*, 41(5):1235–1265, 2012.

31 Jonah Sherman. Nearly Maximum Flows in Nearly Linear Time. In *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS*, pages 263–269, 2013.

Optimal Distributed Covering Algorithms

Ran Ben-Basat
Harvard University, Cambridge, MA, United States
ran@seas.harvard.edu

Guy Even
Tel Aviv University, Tel Aviv, Israel
guy@eng.tau.ac.il

Ken-ichi Kawarabayashi
National Institute of Informatics, Tokyo, Japan
k_keniti@nii.ac.jp

Gregory Schwartzman
National Institute of Informatics, Tokyo, Japan
greg@nii.ac.jp

Abstract

We present a time-optimal deterministic distributed algorithm for approximating a minimum weight vertex cover in hypergraphs of rank f. This problem is equivalent to the Minimum Weight Set Cover problem in which the frequency of every element is bounded by f. The approximation factor of our algorithm is $(f+\varepsilon)$. Let Δ denote the maximum degree in the hypergraph. Our algorithm runs in the CONGEST model and requires $O(\log\Delta/\log\log\Delta)$ rounds, for constants $\varepsilon \in (0,1]$ and $f \in \mathbb{N}^+$. This is the first distributed algorithm for this problem whose running time does not depend on the vertex weights nor the number of vertices. Thus adding another member to the exclusive family of *provably optimal* distributed algorithms.

For constant values of f and ε, our algorithm improves over the $(f+\varepsilon)$-approximation algorithm of [18] whose running time is $O(\log\Delta + \log W)$, where W is the ratio between the largest and smallest vertex weights in the graph. Our algorithm also achieves an f-approximation for the problem in $O(f\log n)$ rounds, improving over the classical result of [15] that achieves a running time of $O(f\log^2 n)$. Finally, for weighted vertex cover ($f=2$) our algorithm achieves a *deterministic* running time of $O(\log n)$, matching the *randomized* previously best result of [16].

We also show that integer covering-programs can be reduced to the Minimum Weight Set Cover problem in the distributed setting. This allows us to achieve an $(f+\varepsilon)$-approximate integral solution in $O\left((1+f/\log n)\cdot\left(\frac{\log\Delta}{\log\log\Delta}+(f\cdot\log M)^{1.01}\cdot\log\varepsilon^{-1}\cdot(\log\Delta)^{0.01}\right)\right)$ rounds, where f bounds the number of variables in a constraint, Δ bounds the number of constraints a variable appears in, and $M=\max\{1,\lceil 1/a_{\min}\rceil\}$, where $a_{\min}$ is the smallest normalized constraint coefficient. This improves over the results of [18] for the integral case, which combined with rounding achieves the same guarantees in $O\left(\varepsilon^{-4}\cdot f^4\cdot\log f\cdot\log(M\cdot\Delta)\right)$ rounds.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Distributed Algorithms, Approximation Algorithms, Vertex Cover, Set Cover

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.5

Related Version `http://arxiv.org/abs/1902.09377`

Funding *Ran Ben-Basat*: Supported by the Zuckerman Institute, the Technion Hiroshi Fujiwara Cyber Security Research center, and the Israel Cyber Directorate.
Ken-ichi Kawarabayashi: Supported by JSPS Kakenhi Grant Number JP18H05291.
Gregory Schwartzman: Supported by JSPS Kakenhi Grant Number JP19K20216 and JP18H05291.

Acknowledgements We thank the anonymous reviewers for their helpful remarks.

© Ran Ben-Basat, Guy Even, Ken-ichi Kawarabayashi, and Gregory Schwartzman;
licensed under Creative Commons License CC-BY

1 Introduction

In the Minimum Weight Hypergraph Vertex Cover (MWHVC) problem, we are given a hypergraph $G = (V, E)$ with vertex weights $w : V \to \{1, \ldots, W\}$.[1] The goal is to find a minimum weight *cover* $U \subseteq V$ such that $\forall e \in E : e \cap U \neq \emptyset$. In this paper we develop a distributed approximation algorithm for MWHVC in the CONGEST model. The approximation ratio is $f + \varepsilon$, where f denotes the rank of the hypergraph (i.e., f is an upper on the size of every hyperedge). The MWHVC problem is a generalization of the Minimum Weight Vertex Cover (MWVC) problem (in which $f = 2$). The MWHVC problem is also equivalent to the Minimum Weight Set Cover Problem (the rank f of the hypergraph corresponds to the maximum frequency of an element). Both of these problems are among the classical NP-hard problems presented in [14].

We consider the following distributed setting for the MWHVC problem. The communication network is a bipartite graph $H(E \cup V, \{\{e, v\} \mid v \in e\})$. We refer to the network vertices as *nodes* and network edges as *links*. The nodes of the network are the hypergraph vertices on one side and hyperedges on the other side. There is a network link between vertex $v \in V$ and hyperedge $e \in E$ iff $v \in e$. The computation is performed in synchronous rounds, where messages are sent between neighbors in the communication network. As for message size, we consider the CONGEST model where message sizes are bounded to $O(\log |V|)$. This is more restrictive than the LOCAL model where message sizes are unbounded.

1.1 Related work

We survey previous results for MWHVC and MWVC. A comprehensive list of previous results appears in Tables 1 and Table 2.

Vertex Cover. The understanding of the round complexity for distributed MWVC has been established in two papers: a lower bound in [19] and a matching upper bound in [4]. Let Δ denote the maximum vertex degree in the graph G. The lower bound states that any distributed constant-factor approximation algorithm requires $\Omega(\log \Delta / \log \log \Delta)$ rounds to terminate. This lower bound holds for every constant approximation ratio, for unweighted graphs and even if the message lengths are not bounded (i.e., LOCAL model) [19]. The matching upper bound is a $(2+\varepsilon)$-approximation distributed algorithm in the CONGEST model, for every $\varepsilon = \Omega(\log \log \Delta / \log \Delta)$.[2] In [16] an $O(\log n)$-round 2-approximation randomized algorithm for weighted graphs in the CONGEST model is given. We note that [16] was the first to achieve this running time with no dependence on W, the maximum weight of the nodes. Recently, Ben-Basat et al. [6] showed an $O(OPT^2 \log OPT)$ rounds algorithm for computing the minimal (unweighted) vertex cover and a $O(OPT)$ rounds for a $(2 + \varepsilon)$-approximation. Here, OPT is the size of the smallest cover and thus these algorithms are adequate when a small solution exists.

Hypergraph Weighted Vertex Cover. For constant values of f, Astrand et al. [2] present an f-approximation algorithm for anonymous networks whose running time is $O(\Delta^2 + \Delta \cdot \log^* W)$. Khuller et al. [15] provide a solution that runs in $O(f \cdot \log \varepsilon^{-1} \cdot \log n)$ rounds in the CONGEST model for any $\varepsilon > 0$ and achieves an $(f + \varepsilon)$-approximation. Setting $\varepsilon = 1/W$ (recall that

[1] Let $n \triangleq |V|$. We assume that $|E| = n^{O(1)}$ and $W = n^{O(1)}$.

[2] Recently, the range of ε for which the runtime is optimal was improved to $\Omega(\log^{-c} \Delta)$ for any $c = O(1)$ [5].

$W = \text{poly}(n)$) results in a f-approximation in $O(f \log^2 n)$-rounds. For constant ε and f values, Kuhn et al. [17, 18] present an $(f + \varepsilon)$-approximation algorithm that terminates in $O(\log \Delta + \log W)$ rounds.

For the Minimum Cardinality (i.e., unweighted) Vertex Cover in Hypergraphs Problem, the lower bound was recently matched by [9] with an $(f + \varepsilon)$-approximation algorithm in the CONGEST model. The round complexity in [9] is $O\left(f/\varepsilon \cdot \frac{\log(f\cdot\Delta)}{\log\log(f\cdot\Delta)}\right)$, which is optimal for constant f and ε.[3] The algorithm in [9] and its analysis is a deterministic version of the randomized maximal independent set algorithm of [10].

1.2 Our contributions

In this paper, we present a deterministic distributed $(f + \varepsilon)$-approximation algorithm for minimum weight vertex cover in f-rank hypergraphs, which completes in

$$O\left(f \cdot \log(f/\varepsilon) + \frac{\log \Delta}{\log\log \Delta} + \min\left\{\log \Delta, f \cdot \log(f/\varepsilon) \cdot (\log \Delta)^{0.001}\right\}\right)$$

rounds in the CONGEST model. For any constants $\varepsilon \in (0, 1)$ and $f \in \mathbb{N}^+$ this implies a running time of $O(\log \Delta/ \log\log \Delta)$, which is optimal according to [19]. This is the first distributed algorithm for this problem whose round complexity does not depend on the node weights nor the number of vertices.

Our algorithm is one of a handful of distributed approximation algorithms for *local* problems which are *provably optimal* [8, 7, 3, 4, 11, 9]. Among these are the classic Cole-Vishkin algorithm [8] for 3-coloring a ring, the more recent results of [3] and [4] for MWVC and Maximum Matching, and the result of [9] for Minimum Cardinality Hypergraph Vertex Cover.

Our algorithm also achieves a *deterministic* f-approximation for the problem in $O(f \log n)$ rounds. This improves over the best known deterministic result for hypergraphs $O(f \log^2 n)$ [15] and matches the best known *randomized* results for weighted vertex cover ($f = 2$) of $O(\log n)$-rounds [16].

We also show that general covering Integer Linear Programs (ILPs) can be reduced to MWHVC in the distributed setting. That is, LP constraints can be translated into hyperedges such that a cover for the hyperedges satisfies all covering constraints. This allows us to achieve an $O\left((1 + f/\log n) \cdot \left(\frac{\log \Delta}{\log\log \Delta} + (f \cdot \log M)^{1.01} \cdot \log \varepsilon^{-1} \cdot (\log \Delta)^{0.01}\right)\right)$ rounds $(f+\varepsilon)$-approximate *integral* solution, where f bounds the number of variables in a constraint, Δ bounds the number of constraints a variable appears in, and $M = \max\{1, \lceil 1/a_{\min}\rceil\}$, where a_{min} is the smallest normalized constraint coefficient. This significantly improves over the results of [18] for the integral case, which combined with rounding achieves the same guarantees in $O\left(\varepsilon^{-4} \cdot f^4 \cdot \log f \cdot \log(M \cdot \Delta)\right)$ rounds. Note that the results of [18] also include a $(1+\varepsilon)$-approximation for the fractional case, while our result only allows for an integral solution. We also note that plugging $\varepsilon = 1/(nWM)$ into our algorithm, achieves an f-approximation for ILPs in polylogarithmic time, a similar result cannot be achieved using [18].

[3] The authors state their result for an $f(1 + \varepsilon)$-approximation which removes the f factor from the runtime.

Table 1 Previous distributed algorithms for MWVC. In the table, $n = |V|$ and $\varepsilon \in (0,1)$. Some of the algorithms hold only for the unweighted case and some are randomized. For randomized algorithms the running times hold in expectation or with high probability.

det.	weighted	approximation	time	algorithm
yes	no	3	$O(\Delta)$	[21]
yes	no	2	$O(\Delta^2)$	[1]
yes	yes	2	$O(1)$ for $\Delta \leq 3$	[1]
yes	yes	2	$O(\Delta + \log^* n)$	[20]
yes	yes	2	$O(\Delta + \log^* W)$	[2]
yes	yes	2	$O(\log^2 n)$	[15]
yes	yes	2	$O(\log n \log \Delta / \log^2 \log \Delta)$	[5]
no	yes	2	$O(\log n)$	[12, 16]
yes	**yes**	**2**	$\boldsymbol{O(\log n)}$	**This work**
yes	yes	$2+\varepsilon$	$O(\varepsilon^{-4} \log(W \cdot \Delta))$	[13, 18]
yes	yes	$2+\varepsilon$	$O(\log \varepsilon^{-1} \log n)$	[15]
yes	yes	$2+\varepsilon$	$O(\varepsilon^{-1} \log \Delta / \log \log \Delta)$	[4]
yes	yes	$2+\varepsilon$	$O\left(\frac{\log \Delta}{\log \log \Delta} + \frac{\log \varepsilon^{-1} \log \Delta}{\log^2 \log \Delta}\right)$	[5]
yes	**yes**	$\boldsymbol{2+\varepsilon}$	$\boldsymbol{O\left(\frac{\log \Delta}{\log \log \Delta} + \log \varepsilon^{-1} \cdot (\log \Delta)^{0.001}\right)}$	**This work**
yes	yes	$2 + \frac{\log \log \Delta}{c \cdot \log \Delta}$	$O(\log \Delta / \log \log \Delta)$	[4], $\forall c = O(1)$
yes	yes	$2 + (\log \Delta)^{-c}$	$O(\log \Delta / \log \log \Delta)$	[5], $\forall c = O(1)$
yes	**yes**	$\boldsymbol{2 + 2^{-c \cdot (\log \Delta)^{0.99}}}$	$\boldsymbol{O(\log \Delta / \log \log \Delta)}$	**This work**, $\boldsymbol{\forall c = O(1)}$

Table 2 Previous distributed algorithms for MWHVC. In the table, $n = |V|$ and $\varepsilon \in (0,1)$. All algorithms are deterministic. Note that [9] holds only for unweighted hypergraphs.

weighted	approximation	time	algorithm
yes	f	$O\left(f^2\Delta^2 + f\Delta \log^* W\right)$	[2]
yes	f	$O\left(f \log^2 n\right)$	[15]
yes	$\boldsymbol{f}$	$\boldsymbol{O\left(f \log n\right)}$	**This work**
no	$f+\varepsilon$	$O\left(\varepsilon^{-1} \cdot f \cdot \frac{\log(f\Delta)}{\log \log(f\Delta)}\right)$	[9]
yes	$f+\varepsilon$	$O\left(f \cdot \log(f/\varepsilon) \cdot \log n\right)$	[15]
yes	$f+\varepsilon$	$O\left(\varepsilon^{-4} \cdot f^4 \cdot \log f \cdot \log(W \cdot \Delta)\right)$	[18]
yes	$\boldsymbol{f+\varepsilon}$	$\boldsymbol{O\left(f \cdot \log(f/\varepsilon) \cdot (\log \Delta)^{0.001} + \frac{\log \Delta}{\log \log \Delta}\right)}$	**This work**
no	$f + 1/c$	$O\left(\log \Delta / \log \log \Delta\right)$	[9], $\forall f, c = O(1)$
yes	$\boldsymbol{f + 2^{-c \cdot (\log \Delta)^{0.99}}}$	$\boldsymbol{O\left(\log \Delta / \log \log \Delta\right)}$	**This work**, $\boldsymbol{\forall f, c = O(1)}$

1.3 Tools and techniques

The Primal-Dual schema. The Primal-Dual approach introduces, for every hyperedge $e \in E$, a dual variable denoted by $\delta(e)$. The dual edge packing constraints are $\forall v \in V, \sum_{v \in e} \delta(e) \leq w(v)$. If for some $\beta \in [0,1)$ it holds that $\sum_{v \in e} \delta(e) \geq (1-\beta) \cdot w(v)$, we say the node v is β-tight. Let $\beta = \varepsilon/(f+\varepsilon)$. For every feasible dual solution, the weight of the set of β-tight vertices is at most $(f+\varepsilon)$ times the weight of an optimal (fractional) solution. The algorithm terminates when the set of β-tight edges constitutes a vertex cover.

The challenge. When designing a Primal-Dual distributed algorithm, the main challenge is in controlling the rate at which we increase the dual variables. On the one hand, we must grow them rapidly to reduce the number of communication rounds. On the other hand, we may not violate the edge packing constraints. This is tricky in the distributed environments as we have to coordinate between nodes. For example, the result of [4] does not generalize to hypergraphs, as hyperedges require the coordination of more than two nodes in order to increment edge variables.

Our algorithm. The algorithm proceeds in iterations, each of which requires a constant number of communication rounds. We initialize the dual variables in a "safe" way so that feasibility is guaranteed. We refer to the additive increase of the dual variable $\delta(e)$ as $\text{bid}(e)$. Similarly to [5] (where bids are called deals), we use *levels* to measure the progress made by a vertex. Whenever the level of a vertex increases, it sends a message about it to all incident edges, which multiply (decrease) their bids by 0.5. Intuitively, the level of a vertex equals the logarithm of its uncovered portion. Formally, we define the level of a vertex v as $\ell(v) \triangleq \left\lfloor \log \frac{w(v)}{w(v)-\sum_{e\ni v}\delta(e)} \right\rfloor$. That is, the initial level of v is 0 and it is increased as the dual variables of the edges incident to v grow. The level of a vertex never reaches $z \triangleq \lceil \log \beta^{-1} \rceil$ as this implies that it is β-tight and entered the cover. Loosely speaking, the algorithm increases the increments $\text{bid}(e)$ exponentially (multiplication by α) provided that no vertex $v \in e$ is $(0.5^{\ell(v)}/\alpha)$-tight with respect to the bids of the previous iteration. Here, $\alpha \geq 2$ is a positive parameter that we determine later. The analysis builds on two observations: (1) The number of times that the increment $\text{bid}(e)$ is multiplied by α is bounded by $\log_\alpha \Delta$. (2) The number of iterations in which $\text{bid}(e)$ is not multiplied by α is bounded by $O(f \cdot z \cdot \alpha)$. Loosely speaking, each such iteration means that for at least one vertex $v \in e$ the sum of bids is at least an $1/(2\alpha)$-fraction of its slack. Therefore, after at most $O(\alpha)$ such iterations that vertex will level up. Since there are z levels per vertex and f vertices in e, we have that the number of such iterations is at most $O(f \cdot z \cdot \alpha)$. Hence the total number of iterations is bounded by $O(\log_\alpha \Delta + f \cdot z \cdot \alpha)$.

Integer linear programs (ILPs). We show distributed reductions that allow us to compute an $(f+\varepsilon)$-approximation for general covering integer linear programs (ILPs). To that end, we first show that any Zero-One covering program (where all variables are binary) can be translated into a set cover instance in which the vertex degree is bounded by 2^f times the bound on the number of constraints each ILP variable appears in. We then generalize to arbitrary covering ILPs by replacing each variable with multiple vertices in the hypergraph, such that the value associated with the ILP variable will be the weighted sum of the vertices in the cover.

2 Problem Formulation

Let $G = (V, E)$ denote a hypergraph. Vertices in V are equipped with positive weights $w(v)$. For a subset $U \subseteq V$, let $w(U) \triangleq \sum_{v\in U} w(v)$. Let $E(U)$ denote the set of hyperedges that are incident to some vertex in U (i.e., $E(U) \triangleq \{e \in E \mid e \cap U \neq \emptyset\}$).

The *Minimum Weight Hypergraph Vertex Cover* Problem (MWHVC) is defined as follows.

Input: Hypergraph $G = (V, E)$ with vertex weights $w(v)$.
Output: A subset $C \subseteq V$ such that $E(C) = E$.
Objective: Minimize $w(C)$.

The MWHVC Problem is equivalent to the Weighted Set Cover Problem. Consider a set system $(X, \mathcal{U})$, where X denotes a set of elements and $\mathcal{U} = \{U_1, \ldots, U_m\}$ denotes a collection of subsets of X. The reduction from the set system $(X, \mathcal{U})$ to a hypergraph $G = (V, E)$ proceeds as follows. The set of vertices is $V \triangleq \{u_1, \ldots, u_m\}$ (one vertex u_i per subset U_i). The set of edges is $E \triangleq \{e_x\}_{x\in X}$ (one hyperedge e_x per element x), where $e_x \triangleq \{u_i : x \in U_i\}$. The weight of vertex u_i equals the weight of the subset U_i. We now detail the setting for computing a distributed $(f+\varepsilon)$-approximation of the problem.

Input. The input is a hypergraph $G = (V, E)$ with non-negative vertex weights $w : V \to \mathbb{N}^+$ and an approximation ratio parameter $\varepsilon \in (0, 1]$. We denote the number of vertices by n, the rank of G by f (i.e., each hyperedge contains at most f vertices), and the maximum degree of G by Δ (i.e., each vertex belongs to at most Δ edges).

Assumptions. We assume that

(i) Vertex weights are polynomial in $n \triangleq |V|$ so that sending a vertex weight requires $O(\log n)$ bits.

(ii) Vertex degrees are polynomial in n (i.e., $|E(v)| = n^{O(1)}$) so that sending a vertex degree requires $O(\log n)$ bits. Since $|E(v)| \le n^f$, this assumption trivially holds for constant f.

(iii) The maximum degree is at least 3 so that $\log \log \Delta > 0$.

Output. The nodes compute a vertex cover $C \subseteq V$. Namely, for every hyperedge $e \in E$, the intersection $e \cap C$ is not empty. The set C is maintained locally in the sense that every vertex v knows whether it belongs to C or not.

Communication Network. The communication network $N(E \cup V, \{\{e, v\} \mid v \in e\})$ is a bipartite graph. There are two types of nodes in the network: *servers* and *clients*. The set of servers is V (the vertex set of G) and the set of clients is E (the hyperedges in G). There is a link (v, e) from server $v \in V$ to a client $e \in E$ if $v \in e$. We note that the degree of the clients is bounded by f and the degree of the servers is bounded by Δ.

Notation.

- We say that an edge e is *covered* by C if $e \cap C \neq \emptyset$.
- Let $E(v) \triangleq \{e \in E \mid v \in e\}$ denote the set of hyperedges that contain v.
- For every vertex v, the algorithm maintains a subset $E'(v) \subseteq E(v)$ that consists of the uncovered hyperedges in $E(v)$ (i.e., $E'(v) = \{e \in E(v) \mid e \cap C = \emptyset\}$).

3 Distributed Approximation Algorithm for MWHVC

3.1 Parameters and Variables

- The algorithm computes an $(f + \varepsilon)$-approximation where $\varepsilon \in (0, 1]$. The parameter β is defined by $\beta \triangleq \varepsilon/(f + \varepsilon)$, where f is the rank of the hypergraph.
- Each vertex v is assigned a level $\ell(v)$ which is a nonnegative integer.
- We denote the dual variables at the end of iteration i by $\delta_i(e)$ (see Appendix A for a description of the dual edge packing linear program). The amount by which $\delta_i(e)$ is increased in iteration i is denoted by $\text{bid}_i(e)$. Namely, $\delta_i(e) = \sum_{j \le i} \text{bid}_j(e)$.
- The parameter $\alpha \ge 2$ determines the factor by which bids are multiplied. We determine its value in the analysis in the following section.

3.2 Algorithm MWHVC

1. Initialization. Set $C \leftarrow \emptyset$. For every vertex v, set level $\ell(v) \leftarrow 0$ and uncovered edges $E'(v) \leftarrow E(v)$.
2. Iteration $i = 0$. Every edge e collects the weight $w(v)$ and degree $|E(v)|$ from every vertex $v \in e$, and sets: $\text{bid}(e) = 0.5 \cdot \min_{v \in e}\{w(v)/|E(v)|\}$. The value $\text{bid}(e)$ is sent to every $v \in e$. The dual variable is set to $\delta(e) \leftarrow \text{bid}(e)$.

3. For $i = 1$ to ∞ do:
   a. Check β-tightness. For every $v \notin C$, if $\sum_{e \in E(v)} \delta(e) \geq (1-\beta)w(v)$, then v joins the cover C, sends a message to every $e \in E'(v)$ that e is covered, and vertex v terminates.
   b. For every uncovered edge e, if e receives a message that it is covered, then it tells all its vertices that e is covered and terminates.
   c. For every vertex $v \notin C$, if it receives a message from e that e is covered, then $E'(v) \leftarrow E'(v) \setminus \{e\}$. If $E'(v) = \emptyset$, then v terminates (without joining the cover).
   d. Increment levels and scale bids.
      For every active (that has not terminated) vertex[4]:
      While $\sum_{e \in E(v)} \delta(e) > w(v)(1 - 0.5^{\ell(v)+1})$ do
      i. $\ell(v) \leftarrow \ell(v) + 1$
      ii. For every $e \in E'(v)$: $\text{bid}(e) \leftarrow 0.5 \cdot \text{bid}(e)$
   e. For every active vertex, if $\sum_{e \in E'(v)} \text{bid}(e) \leq \frac{1}{\alpha} \cdot 0.5^{\ell(v)+1} \cdot w(v)$, then send the message "raise" to every $e \in E'(v)$; otherwise, send the message "stuck" to every $e \in E'(v)$.
   f. For every uncovered edge e, if *all* incoming messages are "raise" $\text{bid}(e) \leftarrow \alpha \cdot \text{bid}(e)$. Send $\text{bid}(e)$ to every $v \in e$, who updates $\delta(e) \leftarrow \delta(e) + \text{bid}(e)$.

- **Termination.** Every vertex v terminates when either $v \in C$ or every edge $e \in E(v)$ is covered (i.e., $E'(v) = \emptyset$). Every edge e terminates when it is covered (i.e., $e \cap C \neq \emptyset$). We say that the algorithm has terminated if all the vertices and edges have terminated.
- **Execution in congest.** See Section B in the Appendix for a discussion of how Algorithm MWHVC is executed in the CONGEST model.

4 Algorithm Analysis

In this section, we analyze the approximation ratio and the running time of the algorithm. Throughout the analysis, we attach an index i to the variables $\text{bid}_i(e), \delta_i(e)$ and $\ell_i(v)$. The indexed variable refers to its value at the end of the i'th iteration.

4.1 Feasibility and Approximation Ratio

The following invariants are satisfied throughout the execution of the algorithm. In the following claim we bound the sum of the bids of edges incident to a vertex.

▷ Claim 1. If $v \notin C$ at the end of iteration i, then $\sum_{e \in E'(v)} \text{bid}_i(e) \leq 0.5^{\ell_i(v)+1} \cdot w(v)$.

Proof. The proof is by induction on i. For $i = 0$, the claim holds because $\ell_0(v) = 0$ and $\text{bid}_0(e) \leq 0.5 \cdot w(v)/|E(v)|$. The induction step, for $i \geq 1$, considers two cases. (A) If v sends a "raise" message in iteration i, then Step 3e implies that $\sum_{e \in E'(v)} \text{bid}_i(e) \leq 0.5^{\ell(v)+1} \cdot w(v)$, as required. (B) Suppose v sends a "stuck" message in iteration i. By Step 3d, $\text{bid}_i(e) \leq 0.5^{\ell_i(v) - \ell_{i-1}(v)} \cdot \text{bid}_{i-1}(e)$ for every $e \in E'(v)$. The induction hypothesis states that $\sum_{e \in E'(v)} \text{bid}_{i-1}(e) \leq 0.5^{\ell_{i-1}(v)+1} \cdot w(v)$. The claim follows by combining these inequalities. ◁

If an edge e is covered in iteration j, then e terminates and $\delta_i(e)$ is not set for $i \geq j$. In this case, we define $\delta_i(e) = \delta_{j-1}(e)$, namely, the last value assigned to a dual variable.

[4] For simplicity of the description, we assume in step 3(d)ii that every $v \in e$ decides whether $\text{bid}(e)$ is halved. In a distributed setting, $\text{bid}(e)$ is halved if there exists a vertex $v \in e$ that requests such a halving. The distributed implementation of this step is further discussed in Appendix B.

▷ Claim 2. For every $i \geq 1$ and every a vertex v that has not terminated the following inequality holds:

$$w(v)(1 - 0.5^{\ell_i(v)}) \leq \sum_{e \in E(v)} \delta_{i-1}(e) \leq (1 - 0.5^{\ell_i(v)+1}) \cdot w(v) . \tag{1}$$

In addition the dual variables $\delta_i(e)$ constitute a feasible edge packing. Namely,

$$\sum_{e \in E(v)} \delta_i(e) \leq w(v) \qquad \text{for every vertex } v \in V,$$

$$\delta_i(e) \geq 0 \qquad \text{for every edge } e \in E.$$

Proof. We prove the claim by induction on the iteration number i. To simplify the proof, we reformulate the statement of the feasibility of the dual variables to $i-1$. We first prove the induction basis for $i = 1$.

Proof of Eq. 1 for $i = 1$. Fix a vertex v. At the end of iteration 0, $\ell_0(v) = 0$ and $0 < \text{bid}_0(e) \leq w(v)/(2|E(v)|)$, for every $e \in E(v)$. Hence $0 < \sum_{e \in E(v)} \text{bid}_0(e) \leq w(v)/2$. Because $\delta_0(e) = \text{bid}_0(e)$, the condition in Step 3d does not hold, and $\ell_1(v) = \ell_0(v) = 0$. We conclude that Eq. 1 holds for $i = 1$.

Proof of feasibility of $\delta_0(e)$ for $i = 1$. Non-negativity follows from the fact that $\delta_0(e) = \text{bid}_0(e) > 0$. The packing constraint for vertex v is satisfied because $\sum_{e \in E(v)} \text{bid}_0(e) \leq w(v)/2$. This completes the proof of the induction basis.

We now prove the induction step assuming that Eq. 1 holds for δ_{i-1}.

Proof of Eq. 1 for $i > 1$. Since v is not in the cover it is also not β-tight. Step 3d in iteration i increases $\ell(v)$ until Eq. 1 holds for i.

Proof of feasibility of $\delta_{i-1}(e)$ for $i > 1$. Consider a vertex v. If v joins C in iteration $i-1$, then $\delta_{i-1} = \delta_{i-2}$, and the packing constraint of v holds by the induction hypothesis. Otherwise, then by $\delta_{i-1}(e) = \delta_{i-2}(e) + \text{bid}_{i-1}(e)$, Claim 1, and the induction hypothesis for Eq. 1, we have

$$\begin{aligned}\sum_{e \in E(v)} \delta_{i-1}(e) &= \sum_{e \in E(v)} (\delta_{i-2}(e) + \text{bid}_{i-1}(e)) \\ &\leq \left(1 - 0.5^{\ell_{i-1}(v)+1} + 0.5^{\ell_{i-1}(v)+1}\right) \cdot w(v) = w(v) .\end{aligned}$$ ◁

Let opt denote the cost of an optimal (fractional) weighted vertex cover of G.

▶ **Corollary 3.** *Upon termination, the approximation ratio of Algorithm* MWHVC *is $f + \varepsilon$.*

Proof. Throughout the algorithm, the set C consists of β-tight vertices. By Claim 20, $w(C) \leq (f + \varepsilon) \cdot \text{opt}$. Upon termination, C constitutes a vertex cover (as an edge only terminate once covered), and the corollary follows. ◀

4.2 Communication Rounds Analysis

In this section, we prove that the number of communication rounds of Algorithm MWHVC is bounded by (where $\gamma > 0$ is a constant, e.g., $\gamma = 0.001$)

$$O\left(f \cdot \log(f/\varepsilon) + \frac{\log \Delta}{\gamma \cdot \log \log \Delta} + \min\left\{\log \Delta, f \cdot \log(f/\varepsilon) \cdot (\log \Delta)^{\gamma}\right\}\right).$$

It suffices to bound the number of iterations because each iteration consists of a constant number of communication rounds.

Let $z \triangleq \lceil \log_2 \frac{1}{\beta} \rceil$. Note that $z = O(\log(f/\varepsilon))$.

▷ Claim 4. The level of every vertex is always less than z.

Proof. Assume that $\ell_i(v) \geq z$. By Eq. 1, $\sum_{e\in E(v)} \delta_{i-1}(e) \geq w(v) \cdot (1 - 2^{-z}) \geq (1 - \beta) \cdot w(v)$. This implies that v is β-tight and joins the cover in Line 3a before $\ell_i(v)$ reaches z. ◁

4.2.1 Raise or Stuck Iterations

▶ **Definition 5** (e-raise and v-stuck iterations). *An iteration $i \geq 1$ is an e-raise iteration if in Line 3f we multiplied $\text{bid}(e)$ by α. An iteration $i \geq 1$ is a v-stuck iteration if v sent the message "stuck" in iteration i.*

Note that if iteration i is a v-stuck iteration and $v \in e$, then $\text{bid}_i(e) \leq \text{bid}_{i-1}(e)$ and i is not an e-raise iteration.

We bound the number of e-raise iterations as follows.

▶ **Lemma 6.** *The number of e-raise iterations is bounded by $\log_\alpha(\Delta \cdot 2^{f\cdot z})$.*

Proof. Let v^* denote a vertex with minimum normalized weight in e (that is, $v^* \in \text{argmin}_{v\in e}\{w(v)/|E(v)|\}$). The first bid satisfies $\text{bid}_0(e) = 0.5 \cdot w(v^*)/|E(v^*)| \geq 0.5 \cdot w(v^*)/\Delta$. By Claim 1, $\text{bid}_i(e) \leq 0.5 \cdot w(v^*)$. The bid is multiplied by α in each e-raise iteration and is halved at most $f \cdot z$ times. The bound on the number of halvings holds because the number of vertices in the edge is bounded by f, and each vertex halves the bid each time its level is incremented. The lemma follows. ◀

We bound the number of v-stuck iterations as follows.

▶ **Lemma 7.** *For every vertex v and level $\ell(v)$, the number of v-stuck iterations is bounded by α.*

Proof. Notice that when v reached the level $\ell(v)$, we had $\sum_{e\in E(v)} \delta(e) \geq w(v)(1 - 0.5^{\ell(v)})$. The number of v-stuck iterations is then bounded by the number of times it can send a "stuck" message without reaching $\sum_{e\in E(v)} \delta(e) > w(v)(1 - 0.5^{\ell(v)+1})$. Indeed, once this inequality holds, the level of v is incremented. Every stuck iteration implies, by Line 3e, that $\sum_{e\in E'(v)} \text{bid}(e) > \frac{1}{\alpha} \cdot 0.5^{\ell(v)+1} \cdot w(v)$. Therefore, we can bound the number of iteration by $\frac{w(v)(1-0.5^{\ell(v)+1}) - w(v)(1-0.5^{\ell(v)})}{\frac{1}{\alpha}\cdot 0.5^{\ell(v)+1}\cdot w(v)} = \alpha$. ◀

4.2.2 Bound on the Number of Rounds

▶ **Theorem 8.** *For every $\alpha \geq 2$, the number of iterations of Algorithm* MWHVC *is*

$$O\left(\log_\alpha(\Delta \cdot 2^{f\cdot z}) + f \cdot z \cdot \alpha\right) = O\left(\log_\alpha \Delta + f \cdot \log(f/\varepsilon) \cdot \alpha\right).$$

Proof. Fix an edge e. We bound the number of iterations until e is covered. Every iteration is either an e-raise iteration or a v-stuck iteration for some $v \in e$. Since e contains at most f vertices, we conclude that the number of iterations is bounded by the number of e-raise iterations plus the sum over $v \in e$ of the number of v-stuck iterations. The theorem follows from Lemmas 6 and 7. ◀

In Theorem 9 we assume that all the vertices know the maximum degree Δ and that $\Delta \geq 3$. The assumption that the maximal degree Δ is known to all vertices is not required. Instead, each hyperedge e can compute a local maximum degree $\Delta(e)$, where $\Delta(e) \triangleq \max_{u\in e} |E(u)|$. The local maximum degree $\Delta(e)$ can be used instead of Δ to define local value of the multiplier $\alpha = \alpha(e)$. Let $T(f, \Delta, \varepsilon)$ denote the round complexity of Algorithm MWHVC. By setting α appropriately, we bound the running time as follows.

▶ **Theorem 9.** [5] *Let $\gamma > 0$ denote a constant and set*

$$\alpha = \begin{cases} \max\left(2, \frac{\log\Delta}{f\cdot\log(f/\varepsilon)\cdot\log\log\Delta}\right) & \text{if } \frac{\log\Delta}{f\cdot\log(f/\varepsilon)\cdot\log\log\Delta} \geq (\log\Delta)^{\gamma/2} \\ 2 & \text{otherwise.} \end{cases}$$

Then, the round complexity of Algorithm MWHVC *satisfies:*

$$T(f,\Delta,\varepsilon) = O\left(f\cdot\log(f/\varepsilon) + \frac{\log\Delta}{\log\log\Delta} + \min\left\{\log\Delta, f\cdot\log(f/\varepsilon)\cdot(\log\Delta)^{\gamma}\right\}\right).$$

Proof. We prove the theorem using a case analysis.

Case $\alpha = \frac{\log\Delta}{f\cdot\log(f/\varepsilon)\cdot\log\log\Delta} \geq 2$ and $\alpha \geq (\log\Delta)^{\gamma/2}$.

This means that the runtime is bounded by $O\left(\log_\alpha\Delta + f\cdot\log(f/\varepsilon)\cdot\alpha\right) = O\left(\frac{\log\Delta}{\gamma\cdot\log\log\Delta}\right)$.

Case $\alpha = 2$ and $2 > \frac{\log\Delta}{f\cdot\log(f/\varepsilon)\cdot\log\log\Delta} \geq (\log\Delta)^{\gamma/2}$.

Notice that in this case $2 > (\log\Delta)^{\gamma/2}$ which implies that Δ is constant. Therefore, the round complexity of our algorithm is bounded by $O\left(\log\Delta + f\cdot\log(f/\varepsilon)\right) = O\left(f\cdot\log(f/\varepsilon)\right)$.

Case $\frac{\log\Delta}{f\cdot\log(f/\varepsilon)\cdot\log\log\Delta} < (\log\Delta)^{\gamma/2}$.

This implies that

$$\begin{aligned}\log\Delta &\leq \min\left\{\log\Delta, f\cdot\log(f/\varepsilon)\cdot(\log\Delta)^{\gamma/2}\cdot\log\log\Delta\right\} \\ &= O\left(\min\left\{\log\Delta, f\cdot\log(f/\varepsilon)\cdot(\log\Delta)^{\gamma}\right\}\right).\end{aligned}$$

Therefore, since $\alpha = 2$ in this case, the runtime is bounded by

$$O\left(\log\Delta + f\cdot\log(f/\varepsilon)\right) = O\left(f\cdot\log(f/\varepsilon) + \min\left\{\log\Delta, f\cdot\log(f/\varepsilon)\cdot(\log\Delta)^{\gamma}\right\}\right).$$ ◀

Let $W \triangleq \max_v w(v)/\min_v w(v)$. By setting $\varepsilon = 1/(nW)$, we conclude the following result for an f-approximation (recall that the vertex degrees and weights are polynomial in n):

▶ **Corollary 10.** *Algorithm* MWHVC *computes an f-approximation in $O(f\log n)$ rounds.*

Additionally, we get the following range of parameters for which the round complexity is optimal:

▶ **Corollary 11.** *Let $f = O\left((\log\Delta)^{0.99}\right)$ and $\varepsilon = (\log\Delta)^{-O(1)}$. Then, Algorithm* MWHVC *computes an $(f+\varepsilon)$-approximation in $O\left(\frac{\log\Delta}{\log\log\Delta}\right)$ rounds.*

For $f = O(1)$ we also get an extension of range of parameters for which the round complexity is optimal. This extension is almost exponential compared to the $\varepsilon = (\log\Delta)^{-O(1)}$ of [5].

▶ **Corollary 12.** *Let $f = O(1)$ and $\varepsilon = 2^{-O\left((\log\Delta)^{0.99}\right)}$. Then our algorithm computes an $(f+\varepsilon)$-approximation and terminates in $O\left(\frac{\log\Delta}{\log\log\Delta}\right)$ rounds.*

---

[5] The statement of the theorem is asymptotic (in Δ). This means that for every constant γ, it holds that either $\log^{\gamma/2}\Delta \geq \log\log\Delta$ or Δ is bounded by a constant (determined by γ) in which case expressions involving Δ can be omitted from the asymptotic expression.

5 Approximation of Covering ILPs

In this section, we present a reduction from solving covering integer linear programs (ILPs) to MWHVC. This reduction implies that one can distributively compute approximate solutions to covering ILPs using a distributed algorithm for MWHVC.

Notation. Let $\mathbb{N}$ denote the set of natural numbers. Let A denote a real $m \times n$ matrix, $\vec{b} \in \mathbb{R}^n$, and $\vec{w} \in \mathbb{R}^n$. Let $LP(A, \vec{b}, \vec{w})$ denote the linear program min $\vec{w}^T \cdot \vec{x}$ subject to $A \cdot \vec{x} \geq \vec{b}$ and $\vec{x} \geq 0$. Let $ILP(A, \vec{b}, \vec{w})$ denote the integer linear program min $\vec{w}^T \cdot \vec{x}$ subject to $A \cdot \vec{x} \geq \vec{b}$ and $\vec{x} \in \mathbb{N}^n$.

▶ **Definition 13.** *The linear program $LP(A, \vec{b}, \vec{w})$ and integer linear program $ILP(A, \vec{b}, \vec{w})$ are* covering programs *if all the components in $A, \vec{b}, \vec{w}$ are non-negative.*

5.1 Distributed Setting

We denote the number of rows of the matrix A by m and the number of columns by n. Let $f(A)$ (resp., $\Delta(A)$) denote the maximum number of nonzero entries in a row (resp., column) of A.

Given a covering ILP, $ILP(A, \vec{b}, \vec{w})$, the communication network $N(ILP)$ over which the ILP is solved is the bipartite graph $N = (X \times C, E)$, where: $X = \{x_j\}_{j \in [n]}$, $C = \{c_i\}_{i \in [m]}$, and $E = \{(x_j, c_i) \mid A_{i,j} \neq 0\}$. We refer to the nodes in X as variable nodes, and to those in C as constraint nodes. Note that the maximum degree of a constraint node is $f(A)$ and the maximum degree of a variable node is $\Delta(A)$.

We assume that the local input of every variable node x_j consists of w_j and the j'th column of A (i.e., $A_{i,j}$, for $i \in [m]$). Every constraint vertex c_i is given the value of b_i as its local input. We assume that these inputs can be represented by $O(\log(nm))$ bits. In $(f+1)$ rounds, every variable node v_j can learn all the components in the rows i of A such that $A_{i,j} \neq 0$ as well as the component b_i.

5.2 Zero-One Covering Programs

The special case in which a variable may be assigned only the values 0 or 1 is called a *zero-one program.* We denote the zero-one covering ILP induced by a matrix A and vectors $\vec{b}$ and $\vec{w}$ by $ZO(A, \vec{b}, \vec{z})$. Every instance of the MWHVC problem is a zero-one program in which the matrix A is the incidence matrix of the hypergraph. The following lemma deals with the converse reduction.

▶ **Lemma 14.** *Every feasible zero-one covering program $ZO(A, \vec{b}, \vec{w})$ can be reduced to an* MWHVC *instance with rank $f' < f(A)$ and degree $\Delta' < 2^{f(A)} \cdot \Delta(A)$.*

Proof. Let A_i denote the i'th row of the matrix A. For a subset $S \subseteq [n]$, let I_S denote the indicator vector of S. Let $\vec{x} \in \{0,1\}^n$ and let $\sigma_i \triangleq \{j \in [n] \mid A_{i,j} \neq 0\}$. Feasibility implies that $A_i \cdot I_{\sigma_i} \geq b_i$, for every row i. Let $\mathcal{S}_i$ denote the set of all subsets $S \subset [n]$ such that the indicator vector I_S does not satisfy the i'th constraint, i.e., $A_i \cdot I_S < b_i$ The i'th constraint is not satisfied, i.e., $A_i \cdot \vec{x} < b_i$, if and if only there exists a set $S \in \mathcal{S}_i$ such that $\vec{x} = I_S$. Hence, $A_i \cdot \vec{x} < \vec{b}$ if and only if the truth value of the following DNF formula is false: $\varphi_i(x) \triangleq \bigvee_{S \in \mathcal{S}_i} \bigwedge_{j \in \sigma_i \setminus S} \text{not}(x_j)$. By De Morgan's law, we obtain that $\text{not}(\varphi_i(x))$ is equivalent to a monotone CNF formula $\psi_i(x)$ such that $\psi_i(x)$ has less than $2^{f(A)}$ clauses, each of which has length less than $f(A)$. We now construct the hypergraph H for the MWHVC instance as follows. For every row i and every $S \in \mathcal{S}$, add the hyperedge $e_{i,S} = \sigma_i \setminus S$. (Feasibility implies that $e_{i,S}$ is not empty.) Given a vertex cover C of the

hypergraph, every hyperedge is stabbed by C, and hence I_C satisfies all the formulae $\psi_i(x)$, where $i \in [m]$. Hence, $A_i \cdot I_C \geq b_i$, for every i. The converse direction holds as well, and the lemma follows. ◀

How does $N(ILP)$ (a bipartite graph with $m+n$ vertices) simulate the execution of MWHVC over the hypergraph H? Each variable node x_j simulates all hyperedges $e_{i,S}$, where $j \in \sigma_i$ and $S \in \mathcal{S}_i$. First, the variable nodes exchange their weights with the variables nodes they share a constraint with in $O(f(A))$ rounds – first every x_j broadcasts its own weight and then each c_i sends all neighbors the weights it received. At each iteration, the variable node sends a raise/stuck message and whether its level was incremented.[6] Notice that the number of rounds required for each such iteration is $O(1 + f(A)/\log n)$ (i.e., constant for $f(A) = O(\log n)$). Each edge node c_i then broadcasts to all vertices two $f(A)$-bit messages that indicate two subsets of vertices of the edge: those that sent a raise message (the complement sent a stuck message) and those that incremented their level. Each variable node x_j knows how to update its bid with every $e_{i,S}$ for which $j \in \sigma_i$ and $S \in \mathcal{S}_i$.

We summarize the complexity of the distributed algorithm for solving a zero-one covering program.

▷ Claim 15. Denoting by $T(f, \Delta, \varepsilon)$ is the running time of Algorithm MWHVC, There exists a distributed CONGEST algorithm for computing an $(f+\varepsilon)$-approximate solution for zero-one covering programs with running time of $O((1 + f(A)/\log n) \cdot T(f(A), 2^{f(A)} \cdot \Delta(A), \varepsilon))$.

5.3 Reduction of Covering ILPs to Zero-One Covering

Consider the covering ILP $ILP(A, \vec{b}, \vec{w})$. We present a reduction of the ILP to a zero-one covering program.

▶ **Definition 16.** *Define* $M(A, \vec{b}) \triangleq \max_j \max_i \left\{ \frac{b_i}{A_{i,j}} \mid A_{i,j} \neq 0 \right\}$.

We abbreviate and write M for $M(A, \vec{b})$ when the context is clear.

▶ **Proposition 17.** *Limiting $\vec{x}$ to the box $[0, M]^n$ does not increase the optimal value of the ILP.*

▷ Claim 18. Every covering ILP $ILP(A, \vec{b}, \vec{w})$ can be solved by a zero-one covering program $ZO(A', \vec{b}, \vec{w'})$, where $f(A') \leq f(A) \cdot \lceil \log_2(M) + 1 \rceil$ and $\Delta(A') = \Delta(A)$.

Proof. Let $B = \lceil \log_2 M \rceil$. Limiting each variable x_j by M means that we can replace x_j by B zero-one variables $\{x_{j,\ell}\}_{\ell=0}^{B-1}$ that correspond to the binary representation of x_j, i.e., $x_j = \sum_{\ell=0}^{B-1} 2^\ell \cdot x_{j,\ell}$. This replacement means that the dimensions of the matrix A' are $m \times n'$, where $n' = n \cdot B$. The j'th column $A^{(j)}$ of A is replaced by B columns, indexed 0 to $B-1$, where the ℓ'th column equals $2^\ell \cdot A^{(j)}$. The vector $\vec{w'}$ is obtained by duplicating and scaling the entries of $\vec{w}$ in the same fashion. ◀

Combining Claims 15 and 18, we obtain the following result.

▶ **Theorem 19.** *There exists a distributed* CONGEST *algorithm for computing an $(f+\varepsilon)$-approximate solution for covering integer linear programs $ILP(A, \vec{b}, \vec{w})$ with running time of $O((1 + f(A)/\log n) \cdot T(f(A) \cdot \log M, 2^{f(A)} \cdot M \cdot \Delta(A), \varepsilon))$, where $T(f, \Delta, \varepsilon)$ is the running time of Algorithm* MWHVC.

[6] One needs to modify the MWHVC algorithm slightly so that, in each iteration, the level of every vertex is increased by at most 1. We defer the details to Appendix C.

Proof. Let f_{HVC}, f_{ZO}, F_{ILP} denote the ranks of the hypergraph vertex cover instance, zero-one covering program, and ILP, respectively. We use the same notation for maximum degrees. The reduction of zero-one programs to MWHVC in Lemma 14 implies that $f_{HVC} \le f_{ZO}$ and $\Delta_{HVC} \le 2^{f_{ZO}} \cdot \Delta_{ZO}$. The reduction of covering ILPs to zero-one programs in Claim 18 implies that $f_{ZO} \le f_{ILP} \cdot (1 + \log M)$ and $\Delta_{ZO} \le \Delta_{ILP}$. The composition of the reductions gives $f_{HVC} \le f_{ILP} \cdot (1+\log M)$ and $\Delta_{HVC} \le 2^{f_{ILP} \cdot (1+\log M)} \cdot \Delta_{ILP} = 2^{F_{ILP}} \cdot 2M \cdot \Delta_{ILP}$ ◀

After some simplifications, the running time of the resulting algorithm for $(f + \varepsilon)$-approximate integer covering linear programs is

$$O\left((1 + f/\log n) \cdot \left(\frac{\log \Delta}{\log\log \Delta} + (f \cdot \log M)^{1.01} \cdot \log \varepsilon^{-1} \cdot (\log \Delta)^{0.01}\right)\right).$$

References

1. Matti Astrand, Patrik Floréen, Valentin Polishchuk, Joel Rybicki, Jukka Suomela, and Jara Uitto. A Local 2-Approximation Algorithm for the Vertex Cover Problem. In *DISC*, 2009.
2. Matti Astrand and Jukka Suomela. Fast distributed approximation algorithms for vertex cover and set cover in anonymous networks. In *SPAA*, 2010.
3. Reuven Bar-Yehuda, Keren Censor-Hillel, Mohsen Ghaffari, and Gregory Schwartzman. Distributed Approximation of Maximum Independent Set and Maximum Matching. In *PODC*. ACM, 2017.
4. Reuven Bar-Yehuda, Keren Censor-Hillel, and Gregory Schwartzman. A Distributed $(2 + \varepsilon)$-Approximation for Vertex Cover in O(log Δ / ε log log Δ) Rounds. *J. ACM*, 2017.
5. R. Ben-Basat, G. Even, K.-i. Kawarabayashi, and G. Schwartzman. A Deterministic Distributed 2-Approximation for Weighted Vertex Cover in $O(\log n \log \Delta / \log^2 \log \Delta)$ Rounds. In *SIROCCO*, 2018.
6. R. Ben-Basat, K.-i. Kawarabayashi, and G. Schwartzman. Distributed Set Cover Approximation: Primal-Dual with Optimal Locality. In *DISC*, 2019.
7. Yi-Jun Chang, Tsvi Kopelowitz, and Seth Pettie. An Exponential Separation between Randomized and Deterministic Complexity in the LOCAL Model. In *FOCS*, 2016.
8. Richard Cole and Uzi Vishkin. Deterministic Coin Tossing with Applications to Optimal Parallel List Ranking. *Information and Control*, 1986.
9. Guy Even, Mohsen Ghaffari, and Moti Medina. Distributed Set Cover Approximation: Primal-Dual with Optimal Locality. In *DISC*, 2018.
10. Mohsen Ghaffari. An improved distributed algorithm for maximal independent set. In *SODA*. Society for Industrial and Applied Mathematics, 2016.
11. Mohsen Ghaffari and Hsin-Hao Su. Distributed Degree Splitting, Edge Coloring, and Orientations. In *SODA*, 2017.
12. Fabrizio Grandoni, Jochen Könemann, and Alessandro Panconesi. Distributed weighted vertex cover via maximal matchings. *ACM Transactions on Algorithms*, 2008. doi:10.1145/1435375.1435381.
13. Dorit S. Hochbaum. Approximation Algorithms for the Set Covering and Vertex Cover Problems. *SIAM J. Comput.*, 1982.
14. Richard M. Karp. Reducibility Among Combinatorial Problems. In *Proceedings of a symposium on the Complexity of Computer Computations*, 1972.
15. Samir Khuller, Uzi Vishkin, and Neal E. Young. A Primal-Dual Parallel Approximation Technique Applied to Weighted Set and Vertex Covers. *J. Algorithms*, 1994.
16. Christos Koufogiannakis and Neal E. Young. Distributed algorithms for covering, packing and maximum weighted matching. *Distributed Computing*, 2011.
17. Fabian Kuhn. *The price of locality: exploring the complexity of distributed coordination primitives*. PhD thesis, ETH Zurich, 2005.

18 Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. The price of being near-sighted. In *SODA*, 2006.
19 Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. Local Computation: Lower and Upper Bounds. *J. ACM*, 2016.
20 Alessandro Panconesi and Romeo Rizzi. Some simple distributed algorithms for sparse networks. *Distributed Computing*, 2001.
21 Valentin Polishchuk and Jukka Suomela. A simple local 3-approximation algorithm for vertex cover. *Inf. Process. Lett.*, 2009.

A Primal-Dual Approach

The fractional LP relaxation of MWHVC is defined as follows.

$$\text{minimize: } \sum_{v\in V} w(v)\cdot x(v)$$

subject to:

$$\sum_{v\in e} x(v) \geq 1, \quad \forall e \in E; \qquad x(v)\geq 0, \quad \forall v \in V \tag{$\mathcal{P}$}$$

The dual LP is an *Edge Packing* problem defined as follows:

$$\text{maximize: } \sum_{e\in E} \delta(e)$$

subject to:

$$\sum_{e\ni v} \delta(e) \leq w(v), \quad \forall v \in V; \qquad \delta(e)\geq 0, \quad \forall e \in E \tag{$\mathcal{D}$}$$

The following claim is used for proving the approximation ratio of the MWHVC algorithm.

▷ Claim 20. Let opt denote the value of an optimal fractional solution of the primal LP ($\mathcal{P}$). Let $\{\delta(e)\}_{e\in E}$ denote a feasible solution of the dual LP ($\mathcal{D}$). Let $\varepsilon \in (0,1)$ and $\beta \triangleq \varepsilon/(f+\varepsilon)$. Define the β-tight vertices by $T_\varepsilon \triangleq \{v\in V \mid \sum_{e\ni v}\delta(e) \geq (1-\beta)\cdot w(v)\}$. Then $w(T_\varepsilon) \leq (f+\varepsilon)\cdot \mathsf{opt}$.

Proof.

$$w(T_\varepsilon) = \sum_{v\in T_\varepsilon} w(v) \qquad \leq \frac{1}{1-\beta}\cdot\left(\sum_{v\in T_\varepsilon}\sum_{e\ni v}\delta(e)\right) \leq \frac{f}{1-\beta}\sum_{e\in E}\delta(e) \leq (f+\varepsilon)\cdot\mathsf{opt}.$$

The last transition follows from $f/(1-\beta) = f+\varepsilon$ and by weak duality. The claim follows. ◁

B Adaptation to the CONGEST model

We need to show that the message lengths in Algorithm MWHVC are $O(\log n)$.

1. In round 0, every vertex v sends its weight $w(v)$ and degree $|E(v)|$ to every hyperedge in $e \in E(v)$. We assume that the weights and degrees are polynomial in n, hence the length of the binary representations of $w(v)$ and $|E(v)|$ is $O(\log n)$.
   Every hyperedge e sends back to every $v\in e$ the pair $(w(v_e), |E(v_e)|)$, where v_e has the smallest normalized weight, i.e., $v_e = \text{argmin}_{v\in e}\{w(v)/|E(v)|\}$.
   Every vertex $v \in e$ locally computes $\text{bid}_0(e) = w(v_e)/(2\cdot|E(v_e)|)$ and $\delta_0(e) = \text{bid}_0(e)$.

2. In round $i \geq 1$, every vertex sends messages. Messages of the sort: "e is covered", "raise", or "stuck" require only a constant number of bits. The increment of v's level needs to be sent to the edges in $E(v)$. These increments require $O(\log z) = O(\log n)$ bits.
3. Every edge e sends to every $v \in e$ the number of times that $\text{bid}(e)$ is halved in this iteration. This message is $O(\log z)$ bits long.
4. Every edge sends the final bid to the vertices. Instead of sending the value of the bid, the edge can send a single bit indicating whether the bid was multiplied by α.
5. Finally, if $\alpha = \alpha(e)$ is set locally based on the local maximum degree $\max_{v \in e} |E(v)|$, then every vertex v sends its degree to all the edges $e \in E(v)$. The local maximum degree for e is sent to every vertex $v \in V$, and this parameter is used to compute $\alpha(e)$ locally.

C Algorithm with At Most One Level Increment per Iteration

We propose to do a single change that will ensure that no vertex levels up more than once per iteration. To that end, we modify Line 3f of our MWHVC algorithm to

- For every uncovered edge e, if *all* incoming messages are "raise" $\text{bid}(e) \leftarrow \alpha \cdot \text{bid}(e)$. Send $\text{bid}(e)$ to every $v \in e$, who updates $\delta(e) \leftarrow \delta(e) + \text{bid}(e)/2$.

That is, the algorithm remains intact except that the dual variables $\delta(e)$ are raised by $\text{bid}(e)/2$ rather than by $\text{bid}(e)$. Intuitively, this guarantees that a vertex's slack does not reduce by more than 50% in each iteration and therefore its level may increase by at most one. We revisit the proof of 2, and, specifically, the *Proof of feasibility of $\delta_{i-1}(e)$ for $i > 1$.*

Proof of feasibility of $\delta_{i-1}(e)$ for $i > 1$. Consider a vertex v. If v joins C in iteration $i-1$, then $\delta_{i-1} = \delta_{i-2}$, and the packing constraint of v holds by the induction hypothesis. If $v \notin C$, then by $\delta_{i-1}(e) = \delta_{i-2}(e) + \text{bid}_{i-1}(e)/2$, Claim 1, and the induction hypothesis for Eq. 1, we have

$$\sum_{e \in E(v)} \delta_{i-1}(e) = \sum_{e \in E(v)} (\delta_{i-2}(e) + \text{bid}_{i-1}(v)/2)$$
$$\leq \left(1 - 0.5^{\ell_{i-1}(v)+1} + 0.5^{\ell_{i-1}(v)+2}\right) \cdot w(v) = \left(1 - 0.5^{(\ell_{i-1}(v)+1)+1}\right) \cdot w(v) \,. \tag{2}$$

As evident by Eq. 2, the vertex v's level may increase by at most once in each iteration.

▶ **Corollary 21.** *For any iteration $i \geq 1$ and vertex v: $\ell_i(v) \leq \ell_{i-1}(v) + 1$.*

We note that the change to the algorithm does not affect the correctness of claims 1, 2, 4 and Lemma 6. Lemma 7 changes slightly, as there can now be twice as many v-stuck iterations:

▶ **Lemma 22.** *For every vertex v and level $\ell(v)$, the number of v-stuck iterations is bounded by 2α.*

Proof. Notice that when v reached the level $\ell(v)$, we had $\sum_{e \in E(v)} \delta(e) \geq w(v)(1 - 0.5^{\ell(v)})$. The number of v-stuck iterations is then bounded by the number of times it can send a "stuck" message without reaching $\sum_{e \in E(v)} \delta(e) > w(v)(1 - 0.5^{\ell(v)+1})$. Indeed, once this inequality holds, the level of v is incremented. Every stuck iteration implies, by Line 3e, that $\sum_{e \in E'(v)} \text{bid}(e) > \frac{1}{\alpha} \cdot 0.5^{\ell(v)+1} \cdot w(v)$. Therefore, we can bound the number of iteration by $\frac{w(v)(1-0.5^{\ell(v)+1}) - w(v)(1-0.5^{\ell(v)})}{\frac{1}{\alpha} \cdot 0.5^{\ell(v)+1} \cdot w(v)/2} = 2\alpha$. ◀

We conclude that the algorithm remains correct and its asymptotic complexity does not change.

Parameterized Distributed Algorithms

Ran Ben-Basat
Harvard University, Cambridge, MA, United States
ran@seas.harvard.edu

Ken-ichi Kawarabayashi
National Institute of Informatics, Tokyo, Japan
k_keniti@nii.ac.jp

Gregory Schwartzman
National Institute of Informatics, Tokyo, Japan
greg@nii.ac.jp

Abstract

In this work, we initiate a thorough study of graph optimization problems parameterized by the *output size* in the distributed setting. In such a problem, an algorithm decides whether a solution of size bounded by k exists and if so, it finds one. We study fundamental problems, including Minimum Vertex Cover (*MVC*), Maximum Independent Set (*MaxIS*), Maximum Matching (*MaxM*), and many others, in both the LOCAL and CONGEST distributed computation models. We present lower bounds for the round complexity of solving parameterized problems in both models, together with optimal and near-optimal upper bounds.

Our results extend beyond the scope of parameterized problems. We show that any LOCAL $(1+\epsilon)$-approximation algorithm for the above problems must take $\Omega(\epsilon^{-1})$ rounds. Joined with the $(\epsilon^{-1}\log n)^{O(1)}$ rounds algorithm of [18] and the $\Omega\left(\sqrt{\frac{\log n}{\log\log n}}\right)$ lower bound of [22], the lower bounds match the upper bound up to polynomial factors in both parameters. We also show that our parameterized approach reduces the runtime of exact and approximate CONGEST algorithms for *MVC* and *MaxM* if the optimal solution is small, without knowing its size beforehand. Finally, we propose the first $o(n^2)$ rounds CONGEST algorithms that approximate *MVC* within a factor strictly smaller than 2.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Distributed Algorithms, Approximation Algorithms, Parameterized Algorithms

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.6

Related Version https://arxiv.org/abs/1807.04900

Funding *Ran Ben-Basat*: Supported by the Zuckerman Institute, the Technion Hiroshi Fujiwara Cyber Security Research center, and the Israel Cyber Directorate.
Ken-ichi Kawarabayashi: Supported by JSPS Kakenhi Grant Number JP18H05291.
Gregory Schwartzman: Supported by JSPS Kakenhi Grant Number JP19K20216 and JP18H05291.

Acknowledgements The authors thank the anonymous reviewers for their helpful remarks. We also thank Keren Censor-Hillel, Guy Even, Seri Khoury, and Ariel Kulik for helpful discussion and comments.

1 Introduction

We initiate the study of distributed algorithms for graph optimization problems parameterized on output size which are fundamental both in the sequential and distributed settings. When we refer to parametrized algorithms in this paper, we consider the output size as the parameter, which is called the *standard parametrization* [27]. Broadly speaking, in these problems we aim to find some underlying graph structure (e.g., a set of nodes) that abides a set of constraints (e.g., cover all edges) while minimizing the cost.

© Ran Ben-Basat, Ken-ichi Kawarabayashi, and Gregory Schwartzman;
licensed under Creative Commons License CC-BY

While parameterized algorithms have received much attention in the sequential setting, not even the most fundamental problems (e.g., Vertex Cover) have a distributed counterpart[1]. We present parameterized upper and lower bounds for fundamental problems such as Minimum Vertex Cover, Maximum Matching and many more.

Motivation. In the sequential setting, some combinatorial optimization problems are known to have polynomial time algorithms (e.g., Maximum Matching), while others are NP-complete (e.g., Minimum Vertex Cover). To deal with the hardness of finding an optimal solution to these problems, the field of parameterized complexity asks what is the best running time we may achieve with respect to some parameter of the problem, instead of the size of the input instance. This parameter, usually denoted by k, is typically taken to be the size of the solution. Thus, even a running time that is exponential in k may be acceptable for small values of k.

In the distributed setting, a network of nodes, which is represented by a communication graph $G(V, E)$, aims to solve some graph problem with respect to G. Computation proceeds in synchronous rounds, in each of which every vertex can send a message to each of its neighbors. The running time of the algorithm is measured as the number of communication rounds it takes to finish. There are two primary communication models: LOCAL, which allows sending messages of unbounded size, and CONGEST, which limits messages to $O(\log n)$ bits (where $n = |V|$).

The above definition implies that the notion of "hardness" in the distributed setting is different. Because we do not take the computation time of the nodes into account, we can solve any problem in $O(D)$ rounds of the LOCAL model (where D is the diameter of the graph), and $O(n^2)$ rounds of the CONGEST model. This is achieved by having every node in the graph learn the entire graph topology and then solve the problem. Indeed, there exist "hard" problems in the distributed setting, where the dependence on D and n is rather large.

There are several lower bounds for distributed combinatorial optimization problems for both the LOCAL and the CONGEST models. For example, [28] provides lower bounds of $\widetilde{\Omega}(\sqrt{n} + D)$ (Where $\widetilde{\Omega}$ hides polylogarithmic factors in n), for a range of problems including MST, min-cut, min s-t cut and many more. There are also $\widetilde{\Omega}(n)$ bounds in the CONGEST model for problems such as approximating the network's diameter [1] and finding weighted all-pairs shortest paths [10]. Recently, the first near-quadratic (in n) lower bounds were shown by [10] for computing exact Vertex Cover and Maximum Independent Set.

The above shows that, similar to the sequential setting, the distributed setting also has many "hard" problems that can benefit from the parameterized complexity lens. Recently, the study of parameterized algorithms for *MVC* and *MaxM* was also initiated in the streaming environment by [11, 12]. This provides further motivation for our work, showing that indeed non-standard models of computation can benefit from parameterized algorithms.

1.1 Our results

Given the above motivation we consider the following fundamental problems (See Section 2 for formal definitions): Minimum Vertex Cover (*MVC*), Maximum Independent Set (*MaxIS*), Minimum Dominating Set (*MDS*), Minimum Feedback Vertex Set (*MFVS*), Maximum Matching (*MaxM*), Minimum Edge Dominating Set (*MEDS*), Minimum Feedback Edge Set (*MFES*). We use $\mathcal{P}$ to denote this problem set.

[1] Many parameters other than the output size have been considered, such as the maximum degree, arboricity, etc.

The problems are considered in both the LOCAL and CONGEST model, where we present lower bounds for the round complexity of solving parameterized problems in both models, together with optimal and near-optimal upper bounds. We also extend existing results [10,22] to the parameterized setting. Some of these extensions are rather direct, but are presented to provide a complete picture of the parameterized distributed complexity landscape.

Our results also extend beyond the scope of parameterized problems. We show that any LOCAL $(1+\epsilon)$-approximation algorithm for the above problems must take $\Omega(\epsilon^{-1})$ rounds. Joined with the $(\epsilon^{-1}\log n)^{O(1)}$ rounds algorithm of [18] and the $\Omega\left(\sqrt{\log n/\log\log n}\right)$ lower bound of [22], the lower bounds match the upper bound up to polynomial factors in both parameters. We also show that our parameterized approach reduces the runtime of exact and approximate CONGEST algorithms for *MVC* and *MaxM* if the optimal solution is small, without knowing its size beforehand. Finally, we propose the first $o(n^2)$ rounds CONGEST algorithms that approximate *MVC* within a factor strictly smaller than 2.

We note that considering parameterized algorithms in the distributed setting presents interesting challenges and unique opportunities compared to the classical sequential environment. In essence, we consider the *communication cost* of solving a parameterized problem on a network. On one hand, we have much more resources (and unlimited computational power), but on the other we need to deal with synchronizations and bandwidth restrictions.

We consider combinatorial minimization (maximization) problems where the size of the solution is upper bounded (lower bounded) by k. A parameterized distributed algorithm is given a parameter k (which is known to all nodes), and must output a solution of size at most k (at least k for maximization problems) if such a solution exists. Otherwise, *all vertices* must output that no such solution exists. A similar definition is given for parameterized approximation problems (see Section 2 for more details). For every $P \in \mathcal{P}$ we denote by k-P its parameterized variant and by k-$\mathcal{P}$ all these problems.

Lower bounds (partially deferred to the full version [7]). We show that the problem of *MVC* can be reduced to *MFVS* and *MFES* via standard reductions which also hold in the CONGEST model. There are no known results for *MFVS* and *MFES* in the distributed setting, so the above reductions, albeit simple, immediately imply that all existing lower bounds for *MVC* also apply for *MFVS* and *MFES*. Using the fact that *MFVS* and *MFES* have a global nature, we can achieve stronger lower bounds for the problems. Specifically, we show that no reasonable approximation can be achieved for *MFVS* and *MFES* in $o(D)$ rounds in the LOCAL model. This is formalized in the following theorem.

▶ **Theorem 1.** *For any $k \in \mathbb{N}^+$, any algorithm that solves MFVS or MFES on a graph with a solution of size k to within an additive error of $O(n/D)$ must take $\Omega(D)$ rounds in the LOCAL model.*

Our main result is a novel lower bound for $(1+\epsilon)$-approximation for all problems in $\mathcal{P}$. Our lower bound states that any $(1+\epsilon)$-approximation (deterministic or randomized) algorithm in the LOCAL model for any problem in $\mathcal{P}$ requires $\Omega(\epsilon^{-1})$ rounds. Usually, lower bounds in the distributed setting are given as a function of the input (size, max degree), and not as a factor of approximation ratio. Our lower bound also applies to *Max-Cut* and *Max-DiCut*, whose parameterized variants are not considered in this paper, and thus are not in $\mathcal{P}$. We state the following theorem.

▶ **Theorem 9.** *For any $\epsilon = \Omega(1/n)$ and $\delta < 1/2 - EXP(-\Omega(n\epsilon))$, any Monte-Carlo LOCAL algorithm that computes a $(1+\epsilon)$-multiplicative approximation with probability at least $1-\delta$ for a problem $P \in \{MVC, MaxM, MaxIS, MDS, MEDS, Max\text{-}DiCut, Max\text{-}Cut, MFVS, MFVS\}$ requires $\Omega(\epsilon^{-1})$ rounds.*

In the full version [7], we show a more involved lower bound construction which extends our lower bound to *Max-k-Cut*[2] where $k = O(1)$. Our lower bound also has implications for non-parameterized algorithms.

The problem of finding a maximum matching in the distributed setting is a fundamental problem in the field that received much attention [2,3,15,25,26]. Despite the existence of a variety of approximation algorithms for the problem, no non-trivial result is known for computing an exact solution for general graphs.

Our lower bounds also have implications for computing a $(1+\epsilon)$-approximation of MVC, *MaxM* and *MaxIS* in the LOCAL model. Combined with the $\Omega\left(\sqrt{\frac{\log n}{\log\log n}}\right)$ lower bound of [22], we can express a lower bound to the problem as $\Omega\left(\epsilon^{-1} + \sqrt{\frac{\log n}{\log\log n}}\right) = (\epsilon^{-1}\log n)^{\Omega(1)}$. Together with the result of [18], which presents an $(\epsilon^{-1}\log n)^{O(1)}$ upper bound, this implies that the complexity of computing a $(1+\epsilon)$-approximation is given by $(\epsilon^{-1}\log n)^{\Theta(1)}$.

Finally, we show a simple and generic way of extending lower bounds to the parameterized setting. The problem with many of the existing lower bounds (e.g., [10,22]) is that the size of the solution is $\tilde{O}(n)$ (linear up to polylogarithmic factors). Thus, it might be the case that if the solution is substantially smaller than the input we might achieve a much faster running time. We show that by simply attaching a large graph to the lower bound graph we can achieve the same lower bounds as a function of k, rather than n. This allows us to restate our $(1+\epsilon)$-approximation lower bound and the bounds of [22] and [10] as a function of $k \ll n$. We show that these lower bounds hold for parameterized problems as defined in this paper.

▶ **Theorem 2.** *There exists a family of graphs $G_{k,n}(V,E)$, such that for any $\epsilon = \Omega(1/k)$ and $\delta < 1/2 - EXP(-\Omega(k\epsilon))$, any Monte-Carlo LOCAL algorithm that computes a $(1+\epsilon)$-multiplicative approximation with probability $1-\delta$ for some k-$P \in k$-$\mathcal{P}$ requires $\Omega(\epsilon^{-1})$ rounds. Here, $n = |V|$ can be arbitrarily larger than k.*

▶ **Theorem 3.** *There exists $\underline{k} \in \mathbb{N}$ such that for any $\underline{k} \le k \le 0.99n$, there exists a family of graphs $G_{k,n}(V,E)$, such that any algorithm that solves k-MVC on $G_{k,n}$ in the CONGEST model requires $\Omega(k^2/\log k \log n)$ rounds, where $n = |V|$ can be arbitrarily larger than k.*

▶ **Theorem 4.** *There exists a family of graphs $G_{k,n}(V,E)$, for sufficiently large k, such that any algorithm that computes a constant approximation for k-MVC, k-MaxM, or k-MaxIS for $G_{k,n}$ in the LOCAL model requires $\Omega\left(\sqrt{\log k/\log\log k}\right)$ rounds, where $n = |V|$ can be arbitrarily larger than k.*

Upper bounds. We first define the family of problems whose optimal solution is lower bounded by the graph diameter (*DLB*, see Section 2 for a formal definition). If the optimal solution size (*OPT*) is small, then for minimization DLB problems we can learn the entire graph in $O(OPT)$ LOCAL rounds. The problem is actually for the case when the optimal solution is large, and all vertices in the graph must output that no k-sized solution exists. Here we introduce an auxiliary result which we use as a building block for all of our algorithms.

▶ **Theorem 12.** *There exists an $O(k)$ rounds deterministic algorithm in the CONGEST model that terminates with all vertices outputting* SMALL *if the diameter is bounded by k, and* LARGE *if the diameter is larger than $2k$. If the diameter is between $k+1$ and $2k$, the vertices answer unanimously, but may return either* SMALL *or* LARGE.

[2] In *Max-k-Cut*, we wish to divide the vertices into k sets $\{A_i\}_{i=1}^k$ such that the number of edges whose endpoints are in different sets is maximized. Notice that this is not a parameterized problem.

Table 1 A summary of our round complexity results for k-*MVC* and k-*MaxM*. All lower bounds hold for randomized algorithms as well as deterministic.

Variant	**Upper Bound**		**Lower Bound**	
	LOCAL	CONGEST	LOCAL	CONGEST
Exact	$O(k)$ $[det.]$	$O\left(k+\frac{k^2\log k}{\log n}\right)$ $[rand.]$ $O\left(k^2\right)$ $[det.]$	$\Omega(k)$	$\Omega\left(k+\frac{k^2}{\log k\log n}\right)$ *k-MVC only
$(1+\epsilon)$-approx. $\forall\epsilon=\Omega(1/k)$			$\Omega\left(\epsilon^{-1}+\sqrt{\frac{\log k}{\log\log k}}\right)$	
$(2-\epsilon)$-approx. $\forall\epsilon\in[1/k,1]$		$O\left(k+\frac{(k\epsilon)^2\log(k\epsilon)}{\log n}\right)$ $[rand.]$ $O(k+(k\epsilon)^2)$ $[det.]$		

Using the above, we can check the diameter, have all vertices reject if it is too large, and otherwise have a leader learn the entire graph in $O(k)$ rounds. As for maximization problems (such as *MaxM* and *MaxIS*) the challenge is somewhat different, as the parameter k does not bound the diameter for legal instances. We first check whether the diameter is at most $2k$ or at least $4k$. If it is bounded by $2k$, we can learn the entire graph. Otherwise, we note that any *maximal* solution has size at least k and is a legal solution to the parameterized problem. Thus, we can efficiently compute a maximal solution by having every node/edge which is a local minimum (according to id) enter the matching/independent set. We repeat this k times and finish (this also works in the CONGEST model). We formalize this in the following theorem.

▶ **Theorem 13.** *There exist $O(k)$ rounds LOCAL algorithms for k-MaxM, k-MaxIS, and any minimization problem k-P for $P\in DLB$.*

Next, we consider the problems of k-*MVC* and k-*MaxM* as case studies for the CONGEST model. We show deterministic upper bound of $O(k^2)$ for k-*MVC* and k-*MaxM* (For k-*MVC* this is near-tight according to Theorem 3). We also note that as the complement of a k-*MVC* is a k-*MaxIS* we have a near-tight upper bound of $O(n-k)^2$ for the problem. This means that if k is large (e.g., $k=n-\log n$) the problem is easy to solve. In the CONGEST model, we first verify that the diameter is indeed small. If it is large, we proceed as we did in the LOCAL model for both problems. For k-*MVC*, we use a standard kernelization procedure to reduce the size of the graph. This is done by adding every node of degree larger than k into the cover. The remaining graph has a bounded diameter and a small number of edges; thus we use a leader node to collect the entire graph. The problem of k-*MaxM* is more challenging, as we do not use existing kernelization techniques. Instead, we introduce a new augmentation-based approach for the parameterized problem.

We then show how with the help of randomization we can achieve a running time of $O(k+k^2\log k/\log n)$ for both problems. Note that for $k\ll n$ this can be a substantial, up to quadratic, improvement. Further, for k-*MVC* it brings our round complexity to within a $O(\log^2 k)$ factor from the lower bound.

Approximations. We consider approximation algorithms, in the CONGEST model, for parameterized *MVC* and *MaxM*. We make non-trivial use of the Fidelity Preserving Transformation framework [14] and simultaneously apply multiple reduction rules that reduce the parameter from k to $O(k\epsilon)$. Using this technique, we derive $(2-\epsilon)$-approximations that run faster than our exact algorithms for any $\epsilon=o(1)$. We summarize our other results in Table 1.

Table 2 Our CONGEST round complexity for deterministic *MVC* and *MaxM*, where ϵ is a positive constant (the actual dependency in ϵ^{-1} is logarithmic). Here, OPT is the size of the optimal solution and is *not known* to the algorithm.

	Exact	$(1+\epsilon)$-approx.	2-approx.	$(2+\epsilon)$-approx.
MVC	$O\left(\min\left\{OPT^2\log OPT, n^2\right\}\right)$	$O\left(OPT^2\right)$	$O\left(\min\left\{OPT\log OPT, \frac{\log n\log\Delta}{\log^2\log\Delta}\right\}\right)$	$O\left(\min\left\{OPT, \frac{\log\Delta}{\log\log\Delta}\right\}\right)$
MaxM			$O\left(\min\left\{OPT\log OPT, \Delta+\log^* n\right\}\right)$	$O\left(\min\left\{OPT, \Delta+\log^* n\right\}\right)$

Applications to non-parameterized algorithms (deferred to the full version [7]). We show that our algorithms can also imply faster non-parameterized algorithms if the optimal solution is small, without needing to know its size. Specifically, we combine our exact and approximation algorithms for parameterized k-*MVC* and k-*MaxM* with doubling and a partial binary search for the value of k. Additionally, our solutions can determine whether to run the existing non-parameterized algorithm or follow the parameterized approach. This results in an algorithm whose runtime is the minimum between current approaches and the number of rounds required for the binary search. Our results are presented in Table 2.

We also present *deterministic* algorithms for *MVC*, *MaxM* in the CONGEST model with an approximation ratio strictly better than 2. Namely, our algorithms terminate in $O(OPT\log OPT) = O(n\log n)$ rounds and provide an approximation ratio of $2 - 1/\sqrt{OPT}$. Here, OPT is the size of the optimal solution and is not known to the algorithm. This is the first non-trivial $(2-\epsilon)$-approximation for *MVC*.[3]

1.2 Related work

Distributed Matching and Covering. Both *MVC* and *MaxM* have received significant attention in the distributed setting. We survey on the results relevant to this paper. We start with existing lower bounds. In [10] a family of graphs of increasing size is presented, such that computing an *MVC* for any graph in the family requires $\Omega(n^2/\log^2 n)$ rounds in the CONGEST model. In [22] a family of graphs is introduced such that any constant approximation for *MVC* requires $\Omega\left(\min\left\{\sqrt{\log n/\log\log n}, \log\Delta/\log\log\Delta\right\}\right)$ rounds in the LOCAL model. Both bounds hold for deterministic and randomized algorithms.

For *MVC*, no non-trivial exact distributed algorithms are known. As for approximations, an optimal (for constant values of ϵ) $(2+\epsilon)$-approximate deterministic algorithm (for the weighted variant) in the CONGEST model running in $O(\epsilon^{-1}\log\Delta/\log\log\Delta)$ rounds is given in [4]. [5] then improved the dependency on ϵ to $O(\log\Delta/\log\log\Delta + \log\epsilon^{-1}\log\Delta/\log^2\log\Delta)$, which also results in a faster 2-approximation algorithm by setting $\epsilon = 1/n$. Recently, this was improved further with a $O(\log n)$ rounds deterministic 2-approximation algorithm [6]. In LOCAL, a randomized $(1+\epsilon)$-approximation in $(\epsilon^{-1}\log n)^{O(1)}$ rounds is due to [18].

For *MaxM*, there are no non-trivial known lower bounds for the exact problem. At the time this paper was first made public, the best lower bound for approximations was due to [22], and no exact non-trivial solution was known for the problem in both the LOCAL and CONGEST models. Independently and simultaneously to our results, [2] show an exact algorithm for *bipartite* graphs running in $O(n\log n)$ rounds in the CONGEST model. They also show an $\tilde{\Omega}(D+\sqrt{n})$ lower bound for computing an $O(1+\Theta(1/\sqrt{n}))$-approximation for unweighted *fractional* matching in the CONGEST model.

[3] When this paper was first made public no better than 2 deterministic approximation for *MaxM* was known in the CONGEST model. Independently and simultaneously [2] provide a deterministic algorithm in the CONGEST model achieving a $(1.5+\epsilon)$-approximation for the problem.

As for approximations, much is known. We survey the results for the unweighted case. An optimal randomized $(1+\epsilon)$-approximation in the CONGEST model, running in $O(\log\Delta/\log\log\Delta)$ rounds for constant ϵ is given in [3]. As for deterministic algorithms, the best known results in the LOCAL model are due to [15] and [20]. In [15] a maximal matching algorithm running in $O(\log^2\Delta\log n)$ rounds and a $(2+\epsilon)$-approximate algorithm running in $O(\log^2\Delta\log 1/\epsilon+\log^* n)$ are given. In [20], a $(1+\epsilon)$-approximation randomized algorithm that runs in $\widetilde{O}(\epsilon^{-3}\log\Delta)$ rounds and a deterministic $\widetilde{O}(\epsilon^{-5}\log^3\Delta+\epsilon^{-1}\log^* n)$ round $(1+\epsilon)$-approximation algorithm are presented. Recently, [2] showed a deterministic algorithm running in $O(\epsilon^{-2}\log(\Delta W)+\epsilon^{-1}(\log^2(\Delta/\epsilon)+\log^* n))$ CONGEST rounds, achieving a $(1+\epsilon)(g/(g-1))$-approximation for weighted maximum matching, where g is the length of the shortest odd cycle in the graph.

Distributed parameterized algorithms. Parameterized distributed algorithms were previously considered for detection problems. Namely, in [21] it was shown the detecting k-paths and trees on k-nodes can be done deterministically in $O(k2^k)$ rounds of the broadcast CONGEST model. Similar results were obtained independently by [13] in the context of distributed property testing [9]. Other distributed algorithms have running times which are parameterized by topological properties of the input graph, see for example [17, 19].

2 Preliminaries

In this paper, we consider the classic and parameterized variants of several popular graph packing and covering problems, in the LOCAL and CONGEST models. A solution to these problems is either a vertex-set or an edge-set. In vertex-set solutions, we require that each vertex will know if it is in the solution or not. For edge-set problems, each vertex must know which of its edges are in the solution, and both end-points of an edge must agree. Computation takes place in synchronous rounds during which each node first receives messages from its neighbors, then perform local computation, and finally send messages to its neighbors. Each of the messages sent to a node neighbor may be different from the others, while the size of messages is unbounded in the LOCAL model or of size $O(\log n)$ in the CONGEST. In both models, the communication graph is identical to the graph on which the problem is solved. That is, two nodes may send messages to each other only if they share an edge. In this paper, we consider the following problems: Minimum Vertex Cover (*MVC*), Maximum Independent Set (*MaxIS*), Minimum Dominating Set (*MDS*), Minimum Feedback Vertex Set (*MFVS*), Maximum Matching (*MaxM*), Minimum Edge Dominating Set (*MEDS*), Minimum Feedback Edge Set (*MFES*). For completeness, we provide formal problem definitions in the full version [7]. We use $\mathcal{P}=\{MVC, MaxM, MaxIS, MDS, MEDS, MFVS, MFES\}$ to denote the above set of problems. Given a parameter k, a parameterized algorithm computes a solution of size bounded by k if such exists; otherwise, the nodes must report that no such solution exists. For a problem $P\in\mathcal{P}$, we denote by k-P (e.g., k-*MVC*) the parameterized variant of the problem. We use k-$\mathcal{P}$ to denote all these problems. We note that all nodes know k when the algorithm starts and that the result may not be the optimal solution to the problem.

▶ **Definition 5.** *An algorithm for a* minimization *(respectively,* maximization*) problem k-P must find a solution of size at most (respectively, at least) k if such exists. If no such solution exists then all nodes must report so when the algorithm terminates.*

We now generalize our definition to account for randomized and approximation algorithm.

▶ **Definition 6.** *For $\alpha \geq 1$, an α-approximation algorithm for a* minimization *(respectively,* maximization*) problem k-P must find a solution of size at most αk (respectively, at least k/α) if a solution of size k exists. Otherwise, all nodes must report that no k-sized solution exists.*

▶ **Definition 7.** *Given $\delta \geq 0$ and $\alpha \geq 1$, an α-approximation Monte Carlo algorithm for a problem k-P terminates with an α-approximate solution to k-P with probability at least $1 - \delta$.*

We now define the notion of *Diameter-Lower-Bounded* (*DLB*) problems. Intuitively, this class contains all problems whose optimal solution size is $\Omega(D)$, which allows efficient algorithms for their parameterized variants. For example, *DLB* includes *MVC*, *MaxIS*, *MaxM*, *MDS*, and *MEDS*, but not *MFVS* and *MFES*. Roughly speaking, these problems admit efficient LOCAL parameterized algorithms as the parameter limits the radius that each node needs to see for solving the problem.

▶ **Definition 8.** *An optimization problem P is in DLB if for any input graph G of diameter D, the size of an optimal solution to P is of size $\Omega(D)$.*

3 Lower Bounds

In this section, we present lower bounds for a large family of classical distributed graph problems. In appendix 3, we generalize the results to randomized algorithms, present additional lower bounds, and show that how classic bounds translate to the parameterized problems.

Here, we provide a construction that implies lower bounds for approximating all problems in $\mathcal{P}$. Our lower bounds dictate that any algorithm that computes a $(1+\epsilon)$-approximation, for $\epsilon = \Omega(1/n)$, requires at least $\Omega(\epsilon^{-1})$ rounds in the LOCAL model, even for randomized algorithms. We note that for all of these problems, no known lower bounds [22, 24] imply superlogarithmic lower bound (which we can get, e.g., for $\epsilon = n^{-2/3}$) was known for approximations. Further, for some problems, such as *MaxM*, no such lower bound is known even for exact solutions in the CONGEST model.

We then generalize our approach and show that even in the parameterized variants of the problems (where the optimal solution is bounded by k), $\Omega(\epsilon^{-1})$ rounds are needed for an arbitrarily large graphs (where $n \gg k$). Our approach is based on the observation that for any x-round algorithm, there exists an input graph of many distinct $\Theta(x)$-long paths, such that the algorithm has an additive error on each of the paths, which accumulates over all paths in the construction. Intuitively, we show that for any set of n node identifiers and any algorithm that takes $o(\epsilon^{-1})$ rounds, it is possible to assign identifiers to nodes such that the approximation ratio would be larger than $1+\epsilon$. Our goal in this section is to prove the following theorem.

▶ **Theorem 9.** *For any $\epsilon = \Omega(1/n)$ and $\delta < 1/2 - EXP(-\Omega(n\epsilon))$, any Monte-Carlo LOCAL algorithm that computes a $(1+\epsilon)$-multiplicative approximation with probability at least $1-\delta$ for a problem $P \in \{MVC, MaxM, MaxIS, MDS, MEDS, Max\text{-}DiCut, Max\text{-}Cut, MFVS, MFVS\}$ requires $\Omega(\epsilon^{-1})$ rounds.*

3.1 Basic Construction

We start with lower bounds for the non-parameterized variants of the problems, where the optimal solution may be of size $\Theta(n)$. For integer parameters $r, \ell > 10$, the graph

$$G_{r,\ell} = (\{v_0\} \cup \{v_{i,j} \mid i \in [r], j \in [\ell+1]\}, \{\{v_0, v_{i,0}\} \mid i \in [r]\} \cup \{\{v_{i,j}, v_{i,j+1}\} \mid i \in [r], j \in [\ell]\})$$

consists of r disjoint paths of length ℓ, whose initial nodes are connected to a central vertex v_0. We also consider the digraph $\vec{G}_{r,\ell}$ in which each edge is oriented away from v_0; i.e.,

$$\vec{G}_{r,\ell} = (\{v_0\} \cup \{v_{i,j} \mid i \in [r], j \in [\ell+1]\}, \{(v_0, v_{i,0}) \mid i \in [r]\} \cup \{(v_{i,j}, v_{i,j+1}) \mid i \in [r], j \in [\ell]\})$$

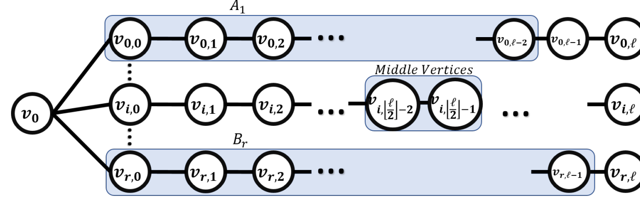


Figure 1 Our basic construction. The middle vertices' ($v_{i,\lfloor\ell/2\rfloor-2}$ and $v_{i,\lfloor\ell/2\rfloor-1}$) output remains the same if we reverse the identifiers along A_i and the algorithm takes less than $\lfloor\ell/2\rfloor-3$ rounds. The output of $v_{i,\lfloor\ell/2\rfloor}$ also remains the same if B_i is flipped. However, since the distance of these vertices from v_0 change, the output is suboptimal for at least one of the orderings.

We present our construction in Figure 1. $G_{r,\ell}$ has $n = r(\ell+1)+1$ vertices and a diameter (as $r > 1$) of 2ℓ. Observe that the optimal solutions of *MVC*, *MaxM*, *MaxIS*, *MDS*, and *MEDS* on $G_{r,\ell}$ (for $r, \ell \geq 3$) have values of $\Theta(r\ell) = \Theta(n)$. For every path $i \in [r]$, let $A_i = \langle v_{i,0}, \ldots, v_{i,\ell-2}\rangle$ and $B_i = \langle v_{i,0}, \ldots, v_{i,\ell-1}\rangle$ denote the longest odd and even length subpaths that do not include v_0 and $v_{i,\ell}$. Given a path P of vertices with assigned identifiers, we denote by P^R a reversal in the order of identifiers. For example, if the identifiers assigned to A_0 were $\langle 0, \ldots, \ell-2\rangle$, then those of A_0^R would be $\langle \ell-2, \ell-3, \ldots, 0\rangle$. This reversal of identifiers along a path plays a crucial role in our lower bounds. Intuitively, if the number of rounds is less than $\ell/2-3$ and we reverse A_i or B_i, the output of the middle vertices *would change* to reflect the mirror image they observe. We show that this implies that on either the original identifier assignment or its reversal, the algorithm must find a sub-optimal solution to the i'th path (where the choice of whether to flip A_i or B_i depends if the output is a vertex set or edge set). In turn, this would sum up to a solution that is far from the optimum by at least an r-additive factor. As the optimal solution is of size $\Theta(r\ell)$, this implies a multiplicative error of $1+\Theta(r/(r\ell)) = 1+1/\ell$.

We show that for arbitrarily large graphs with an optimal solution of size $\Theta(n)$, any x-round algorithm must have an additive error of $\Omega(n/x)$.

▶ **Lemma 10.** *Let $x, r \in \mathbb{N}^+$ be integers larger than 10, and let $\mathcal{I}$ be a set of $n = (2x+4)r+1$ node identifiers. For any deterministic LOCAL algorithm for MVC, MaxM, MaxIS, MDS, MEDS, Max-DiCut, or Max-Cut that terminates in x rounds on $G_{r,2x+3}$, there exists an assignment of vertex identifiers for which the algorithm has an additive error of $\Omega(n/x)$.*

Proof. First, let us characterize the optimal solutions for each of the problems on $G_{r,2x+3}$ (or $\vec{G}_{r,2x+3}$ for *Max-DiCut*). For simplicity of presentation, we assume that $\mathcal{I} = [n]$ and (x mod 6) = 0 although the result holds for any $\mathcal{I}$ and x. We have

$$OPT_{MVC} = \{v_{i,2j} \mid i \in [r], j \in [x+2]\},$$
$$OPT_{MM} = \{\{v_{i,2j}, v_{i,2j+1}\} \mid i \in [r], j \in [x+2]\},$$
$$OPT_{MaxIS} = OPT_{Max\text{-}Cut} = OPT_{Max\text{-}DiCut} = \{v_0\} \cup \{v_{i,2j+1} \mid i \in [r], j \in [x+2]\},^4$$
$$OPT_{MDS} = \{v_{i,3j} \mid i \in [r], j \in [2x/3+1]\},$$
$$OPT_{MEDS} = \{\{v_0, v_{i,0}\} \mid i \in [r]\} \cup \{\{v_{i,3j+2}, v_{i,3j+3}\} \mid i \in [r], j \in [2x/3]\}.^5$$

[4] For *Max-Cut*, the complement solution $V \setminus \{v_0\} \cup \{v_{i,2j+1} \mid i \in [r], j \in [x+2]\}$ is also optimal, but the correctness would follow from similar arguments.

[5] Each edge of the form $\{v_0, v_{i,0}\}$ (for $i \in [r]$) may each be replaced by the $\{v_{i,0}, v_{i,1}\}$ edge. Unlike the other problems, the optimal solution here is not unique, but this does not affect the proof.

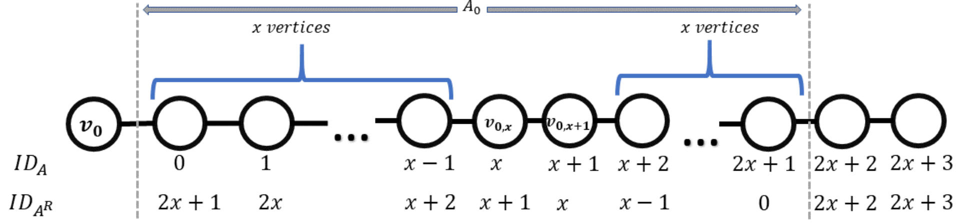


Figure 2 Before the reversal (in A_0), vertices $v_{0,x}$ and $v_{0,x+1}$ have identifiers x and $x+1$. When reversing A_0, the vertices switch identifiers and must switch their output. That is, $out_A(v_{0,x}) = out_{A^R}(v_{0,x+1})$ and $out_A(v_{0,x+1}) = out_{A^R}(v_{0,x})$ in any algorithm that takes fewer than x communication rounds.

For example, this means that the only optimal solution to *MVC* picks all vertices whose distance from v_0 is odd. Next, consider a path $i \in [r]$ and consider the case where every vertex $v_{i,j}$ has identifier $(2x+4)i+j$. From the point of view of the node with identifier $(2x+4)i+x$ (which is $v_{i,x}$ in this assignment), in its x-hop neighborhood it has nodes with identifiers $(2x+4)i, (2x+4)i+1, \ldots, (2x+4)i+x-1$ on one port (side) and identifiers $(2x+4)i+x+1, (2x+4)i+x+2, \ldots, (2x+4)i+2x+2$ on the other. On the other hand, if we reverse A_i (i.e., assign $v_{i,j} \in A_i$ with identifier $(2x+4)i+2x+1-j$), the view of $v_{i,x}$ remains exactly the same. That is, the node observes the exact same topology and vertex identifiers in both cases. Since the algorithm is deterministic, the output of $(2x+4)i+x$ must remain the same for both identifier assignments, even though now it is placed in $v_{i,x+1}$! Similarly, reversing A_i would mean that the node with identifier $(2x+4)i+x+1$ (which changes places from $v_{i,x+1}$ to $v_{i,x}$ after the reversal) also provides the same output in both cases. Therefore, reversing A_i would switch the outputs of $v_{i,x+1}$ and $v_{i,x}$. This implies that the output of the algorithm is suboptimal for either A_i or A_i^R for *MVC*, *MaxM*, *MaxIS*, *Max-Cut*, and *Max-DiCut*. We illustrate this reversal on path 0 in Figure 2. Repeating this argument for B_i, we get that its reversal would switch the outputs of $v_{i,x}$ and $v_{i,x+2}$, making the algorithm err in *MDS* (as $v_{i,x}$ is in the optimal cover while $v_{i,x+2}$ is not).

For MEDS, every u, v that share an edge must agree whether it is in the solution or not. In an optimal solution the edge $\{v_{i,x}, v_{i,x+1}\}$ must be in the dominating set while $\{v_{i,x+1}, v_{i,x+2}\}$ must not. However, by reversing B_i the identifiers of $v_{i,x}$ and $v_{i,x+2}$ switch, changing edge added from $\{v_{i,x}, v_{i,x+1}\}$ to $\{v_{i,x+1}, v_{i,x+2}\}$ or vice versa, implying an error for B_i or B_i^R.

As we showed that there exists an identifier assignment that "fools" the algorithm on every path $i \in [r]$, we conclude that the algorithm has an additive error of at least $r = (n-1)/(2x+4) = \Omega(n/x)$. ◀

Lower bounds for MFVS and MFES follow from the reduction to *MVC* which is presented in the full version [7]. Since the optimal solution to all problems on $G_{r,2x+4}$ is of size $\Theta(n)$, we have that the algorithms have an approximation ratio of $1+\Theta((n/x)/n) = 1+\Theta(1/x)$. Plugging $\epsilon = \Theta(1/x)$ we conclude the following.

▶ **Corollary 11.** *For any $\epsilon = \Omega(1/n)$, any deterministic LOCAL algorithm that computes a $(1+\epsilon)$-multiplicative approximation for any $P \in \mathcal{P}$ requires $\Omega(\epsilon^{-1})$ rounds.*

4 Upper Bound Warmup – Parameterized Diameter Approximation

In this section, we illustrate the concept of parameterized algorithms with the classic problem of diameter approximation. This procedure will also play an important role in all our algorithms. Computing the exact diameter of a graph in the CONGEST model is costly. Specifically, it is known that computing a $(3/2-\epsilon)$-approximation of the diameter, or even distinguishing between diameter 3 or 4, requires $\widetilde{\Omega}(n)$ CONGEST rounds[6] [1, 8, 16, 23]. Computing a 2-approximation for the diameter is straightforward in $O(D)$ rounds, by finding the depth of any shortest paths tree. However, we wish to devise algorithms whose round complexity is bounded by some function $f(k)$, even if no solution of size k exists. Therefore, we now show that it is possible to compute a 2-approximation for the *parameterized version* of the diameter computation problem.

▶ **Theorem 12.** *There exists an $O(k)$ rounds deterministic algorithm in the CONGEST model that terminates with all vertices outputting* SMALL *if the diameter is bounded by k, and* LARGE *if the diameter is larger than $2k$. If the diameter is between $k+1$ and $2k$, the vertices answer unanimously, but may return either* SMALL *or* LARGE*.*

Proof. Our algorithm starts with k rounds, such that in every round each vertex broadcasts the minimal identifier it has learned about (initially its own identifier). After this stage terminates, each vertex v has learned the minimal identifier x_v in its k-hop neighborhood.

Next follows $2k+1$ rounds such that in each round each vertex v broadcasts y_v and z_v, which are the minimal and maximal x_u identifier it has seen so far. That is, we start with $y_v = z_v = x_v$ and at each round we set $y_v \leftarrow \min\{y_v, \min\{y_u \mid u \in N(v)\}\}$ and $z_v \leftarrow \max\{z_v, \max\{z_u \mid u \in N(v)\}\}$. When this ends, each vertex returns *SMALL* if $y_v = z_v$ and *LARGE* otherwise. Clearly, the entire execution takes $O(k)$ rounds.

For correctness, observe that if the diameter is bounded by k then all x_v's are identical to the globally minimal identifier. Next, assume that the diameter is at least $2k+1$, and fix some vertex v. This means that there exist a vertex u whose distance is exactly $k+1$ with respect to x_v, and thus at most $2k+1$ from v. Since the first stage of the algorithm runs for k rounds, we have that $x_u \neq x_v$. Therefore, after $k+1$ rounds of the second stage we have that $y_{x_v} \neq z_{x_v}$, and after additional k rounds $y_v \neq z_v$ and thus v outputs *LARGE*. Finally, if the diameter is between $k+1$ and $2k$, then all vertices have the same y_v and z_v values and thus answer unanimously. ◀

5 Parameterized Problems Upper Bounds

5.1 LOCAL Algorithms

Our first result is for diameter lower bounded problem in the LOCAL model. We show that any minimization problem k-$P \in DLB$ can be solved in $O(k)$ LOCAL rounds. To that end, we first use Theorem 12 to check whether the diameter is at most ck, or at least $2ck$, where c is a constant such that a diameter of at least ck implies that no solution of size k exists. If the diameter is larger than $2ck$, the algorithm terminates and reports that no k-sized solution exists. Otherwise, we collect the entire graph at a leader vertex v which computes the optimal solution. If the solution is of size at most k, v sends it to all vertices. If no solution of size k exists, v notifies the other vertices.

[6] Where $\widetilde{\Omega}(n)$ hides factors polylogarithmic in n.

The above approach does not necessarily work for maximization problems as the existence of a k-sized solution does not imply a bounded diameter. Nevertheless, we now show that k-*MaxM* and k-*MaxIS* have $O(k)$ rounds algorithms. For this, we first check whether the diameter is at most $2k$ or at least $4k$. If the diameter is small, we can still collect the graph and solve the problem locally. Otherwise, we use the fact that *any* maximal matching or Independent Set in a graph with a diameter larger than $2k$ must be of size at least k. Since the maximal matching or independent set may be too large, we run just k iterations of extending the solution. For k-*MaxM*, at each iteration, any edge that neither of its endpoints is matched and that is a local minimum (with respect to the identifiers of its endpoints) joins the matching. We are guaranteed that the matching grows by at least a single edge at each round and thus after k iterations the algorithm terminates. Similarly, for k-*MaxIS*, at each iteration, every vertex that neither of its endpoints is in the independent set and is a local minimum enters the set. We summarize this in the following theorem.

▶ **Theorem 13.** *There exist $O(k)$ rounds LOCAL algorithms for k-MaxM, k-MaxIS, and any minimization problem k-P for $P \in DLB$.*

5.2 CONGEST algorithms for k-MVC

Here we state our results for k-*MVC* in the CONGEST model. Our first algorithm is deterministic and aims to solve the exact variant of k-*MVC*. Intuitively, it works in two phases; first, it checks that the diameter is $O(k)$, if not the algorithm rejects. Knowing that the diameter is bounded by $O(k)$, we proceed by calculating a solution *assuming there exists a solution of size at most k*. If this assumption holds, we are guaranteed to find such a solution. We run the above for just enough rounds to guarantee that if a k-sized solution exists we will find such a solution. Finally, we check that the size of the solution returned by the algorithm is indeed bounded by k.

We first show a procedure that solves the problem *if* a solution of size k exists. If no such solution exists, this procedure may not terminate in time or compute a cover larger than k.

▶ **Lemma 14.** *There exists a deterministic algorithm that if a k-sized cover exists: (1) terminates in $O(k^2)$ CONGEST rounds and (2) finds such a cover.*

Proof. Given that there exists a k-sized cover, the diameter of the graph is bounded by $2k$. Therefore, we can compute a unique leader and a BFS tree rooted at that leader in $O(k)$ rounds. Our first observation is that every v with a degree larger than k must be in any k-sized cover. Thus, every such vertex immediately goes into the cover and gets removed together with all of its adjacent edges. If a vertex has degree 0, it terminates (without entering the cover). Denote the remaining graph by $G' = (V', E')$.

For our analysis, let us fix some vertex cover $X \subseteq V'$ of size k and denote the remaining vertices by $A = V' \setminus X$. We note that the set A is an independent set. Thus, all edges in the graph are either between vertices in X or between A and X. We note that $|X| \leq k$, and now we aim to bound the number of remaining edges. We now show that $|E'| \leq k^2$.

As all vertices with degrees greater than k have been added to the cover and removed, all remaining vertices have a degree of at most k. Because all remaining edges in the graph are of the form $(x, v) \in E', x \in X, v \in A$ or $(u, v) \in E', u, v \in A$, we may immediately bound the number of remaining edges, $|E'| \leq k^2$. We now can just learn the entire graph in time $O(|E'| + |D|) = O(|E'|) = O(k^2)$ using pipelining. The leader vertex computes the optimal cover for G' and notifies all vertices in G' whether they should join it. ◀

▶ **Theorem 15.** *There exists a deterministic algorithm for k-MVC that terminates in $O(k^2)$ rounds.*

Proof. Our algorithm first uses Theorem 12 to estimate the diameter. Specifically, we first apply it for $k' = 2k$. If the vertices report LARGE, we follow the same approach as in the algorithm in the LOCAL model and reject. Otherwise, we proceed knowing the diameter is bounded by $4k$. For the case the diameter is bounded we can compute a unique leader and a BFS tree rooted at that leader in $O(k)$ rounds.

Let $c = O(1)$ such that the algorithm provided in Lemma 14 is guaranteed to terminate in ck^2 rounds for *any* graph G with a cover of size k. We run the algorithm for ck^2 rounds; if the procedure did not terminate, all vertices report that no k-sized cover exists. Finally, we count the number of nodes in the cover using the BFS tree and verify that indeed the size of the solution in G is bounded by k. If any node in the tree sees more than k identifiers of vertices that joined the cover, it notifies all vertices that the solution is invalid and thus no k-sized solution exists. ◀

A Randomized Algorithm. While we show a deterministic LOCAL algorithm for k-MVC that is optimal even if randomization is allowed, we have a gap of $\Theta(\min\{k, \log k \log n\})$ in our CONGEST round complexity. We now present a randomized algorithm with a $O\left(k + \frac{k^2 \log k}{\log n}\right)$ round complexity, thereby reducing the gap to $O(\log^2 k)$. This is achieved by the observation that while node identifiers are of length $\Theta(\log n)$, we can replace each node identifier with an $O(\log k)$-bit *fingerprint*. If there exists a cover of size k, there are at most $k + k^2 \le 2k^2$ vertices in G' (after our reduction rule), and we can use ($\mathfrak{b} = (c+4)\log k + 1$)-bit fingerprints, for some $c > 0$, and get that the probability of collision (that two vertices have the same fingerprint) is at most $\binom{2k^2}{2}2^{-\mathfrak{b}} < k^4 2^{-\mathfrak{b}+1} = k^{-c}$. Next, we run our deterministic algorithm, where each vertex considers its fingerprint as an identifier. Observe that since $|E'| \le k^2$ and each edge encoding now requires $O(\log k)$ bits (for $c = O(1)$), the overall amount of bits sent to the leader is $O(|E'| \log k) = O(k^2 \log k)$. Since the diameter of the graph is $O(k)$, and $O(\log n)$ bits may be transmitted on every round on each edge, we use pipelining to get the round complexity in Theorem 16. Note that we only use fingerprints for the part of the algorithm which requires time quadratic in k. That is, checking the size of the diameter and validating the size of the solution are still done using the original identifiers.

Finally, if no cover of size k exists, we may have more than $2k^2$ vertices in G' and may get fingerprint collisions. Nevertheless, when using the BFS tree to send the (fingerprinted) identifiers to the leader, we can identify that such a collision happened and the leader can notify all vertices to report that no k-sized cover exists.

▶ **Theorem 16.** *For any $\delta = k^{-O(1)}$, there exists a randomized algorithm for k-MVC that terminates in $O\left(k + \frac{k^2 \log k}{\log n}\right)$ rounds, while being correct with probability at least $1 - \delta$.*

Approximations. As we may add all nodes of degree more than k to the cover, this bounds Δ, the maximum degree in the remaining graph, by k. We can now apply the algorithm of [5] which runs in $O(\log \Delta / \log \log \Delta + \log \epsilon^{-1} \log \Delta / \log^2 \log \Delta)$ and achieves a $(2 + \epsilon)$-approximation. This immediately results in a deterministic $O(\log k / \log \log k + \log \epsilon^{-1} \log k / \log^2 \log k)$-round $(2 + \epsilon)$-approximation algorithm in the CONGEST model. Further, since there exists a cover of size $OPT \le k$, setting $\epsilon = 1/(k + 1)$ implies that the resulting cover is of size $\lfloor OPT(2+\epsilon) \rfloor \le 2OPT + \lfloor k/(k+1) \rfloor = 2OPT$. Thus, we conclude that

our algorithm computes a 2-approximation for the problem in $O(\log^2 k/\log^2\log k)$ rounds. Unfortunately, while this succeeds if there indeed exists a solution of size k, validating the size of the solution takes $O(k)$ rounds.

We now expand the discussion and propose an algorithm that computes a $(2-\epsilon)$-approximation. For $\epsilon = o(1)$, this gives a better round complexity than our exact algorithm, while for $\epsilon = O(1/\sqrt{k})$ it improves the approximation ratio of the above 2-approximation while still terminating in $O(k)$ rounds. In the full version [7], we use this algorithm to derive the first $(2-\epsilon)$-approximation algorithm for the (non-parametric) *MVC* problem that runs in $o(n^2)$ rounds.

▶ **Theorem 17.** *$\forall\epsilon \in [1/k, 1]$, there exists a deterministic CONGEST algorithm for k-MVC that computes a $(2-\epsilon)$-approximation in $O\left(k + (k\epsilon)^2\right)$ rounds. For any $\delta = (k\epsilon)^{-O(1)}$, there also exists a randomized algorithm that terminates in $O\left(k + \frac{(k\epsilon)^2 \log(k\epsilon)}{\log n}\right)$ rounds and errs with probability $\leq \delta$.*

Proof. We utilize the framework of Fidelity Preserving Transformations [14]. Intuitively, if there exists a vertex cover of size k in the original graph, and we remove two vertices u, v that share an edge, then the new graph must have a cover of size at most $k-1$. This allows us to reduce the parameter at the cost of introducing error (we add both u and v to the cover, while an optimal solution may only include one of them). This process is called $(1,1)$*-reduction step* as it reduces the parameter by 1 and increases the size of the cover (compared with an optimal solution) by at most 1. Roughly speaking, we repeat the process until the parameter reduces to $k\epsilon$, at which point we run an exact algorithm on the remaining graph.

[14] proved that for any $\alpha \leq 2$, repeating a $(1,1)$-reduction step until the parameter reduces to $k(2-\alpha)$ allows computing an α-approximation by finding an exact solution to the resulting subgraph and adding all vertices that have an edge that was reduced in the process. For our purpose, we set $\alpha = 2-\epsilon$; thus, the exact algorithms only need to find a cover of size $k\epsilon$.

Our algorithm begins by checking the diameter is $O(k)$ and finding a leader vertex v. This is doable in $O(k)$ rounds as having a vertex cover of size k guarantees that the diameter is $O(k)$. We proceed with applying the $(1,1)$-reduction steps. To that end, we compute a maximal matching M and send it to v, which requires $O(k)$ rounds. If $|M| \leq k(1-\epsilon)$, v instructs all matched vertices to enter the cover, and the algorithm terminates with a solution of size at most $2k(1-\epsilon) < k(2-\epsilon)$, as needed. If $|M| > k(1-\epsilon)$, the leader selects an arbitrary submatching $M' \subset M$ of size $k(1-\epsilon)$ and the reduction rules are simultaneously applied for every $e \in M'$. The remaining graph has a cover of size at most $k\epsilon$, at which point we apply the exact algorithms. Finally, we validate the size of the solution as in the above algorithms. By Theorems 15 and 16, we establish the correctness and runtime of our algorithms. ◀

5.3 CONGEST algorithms for *k-MaxM*

In the full version [7], we present deterministic and randomized upper bounds for the exact and approximate variants of the *k-MaxM* matching problem. Our exact algorithm first checks that the diameter is $O(k)$ and then uses a leader node to iteratively compute augmenting paths which extend the matching until it becomes of size k. Our approximations are derived from the Fidelity Preserving Transformations framework [14] and leverage the fact that a maximal matching (or a matching of size k) can be found in $O(k)$ rounds. We then sends the maximal matching to a leader node which arbitrarily selects a $(k/2 - O(k\epsilon))$-sized submatching and sends it to all nodes which then find the exact maximum matching on the residual graph.

References

1 Amir Abboud, Keren Censor-Hillel, and Seri Khoury. Near-Linear Lower Bounds for Distributed Distance Computations, Even in Sparse Networks. In *DISC*, 2016.

2 Mohamad Ahmadi, Fabian Kuhn, and Rotem Oshman. Distributed Approximate Maximum Matching in the CONGEST Model. In *DISC*, 2018.

3 R. Bar-Yehuda, K. Censor-Hillel, M. Ghaffari, and G. Schwartzman. Distributed Approximation of Maximum Independent Set and Maximum Matching. In *PODC*, 2017.

4 Reuven Bar-Yehuda, Keren Censor-Hillel, and Gregory Schwartzman. A Distributed $(2+\epsilon)$-Approximation for Vertex Cover in O(log Δ / ϵ log log Δ) Rounds. *J. ACM*, 2017.

5 R. Ben-Basat, G. Even, K.-i. Kawarabayashi, and G. Schwartzman. A Deterministic Distributed 2-Approximation for Weighted Vertex Cover in $O(\log n \log \Delta / \log^2 \log \Delta)$ Rounds. In *SIROCCO*, 2018.

6 R. Ben-Basat, G. Even, K.-i. Kawarabayashi, and G. Schwartzman. Optimal Distributed Covering Algorithms. In *DISC*, 2019.

7 Ran Ben-Basat, Ken-ichi Kawarabayashi, and Gregory Schwartzman. Parameterized Distributed Algorithms. `arXiv:1807.04900`.

8 Karl Bringmann and Sebastian Krinninger. A Note on Hardness of Diameter Approximation. *Information Processing Letters*, 2018.

9 Keren Censor-Hillel, Eldar Fischer, Gregory Schwartzman, and Yadu Vasudev. Fast Distributed Algorithms for Testing Graph Properties. In *DISC*, 2016.

10 Keren Censor-Hillel, Seri Khoury, and Ami Paz. Quadratic and Near-Quadratic Lower Bounds for the CONGEST Model. In *DISC*, 2017.

11 R. Chitnis, G. Cormode, H. Esfandiari, M. Hajiaghayi, A. McGregor, M. Monemizadeh, and S. Vorotnikova. Kernelization via Sampling with Applications to Finding Matchings and Related Problems in Dynamic Graph Streams. In *SODA*, 2016.

12 R. H. Chitnis, G. Cormode, M. T. Hajiaghayi, and M. Monemizadeh. Parameterized Streaming: Maximal Matching and Vertex Cover. In *SODA*, 2015.

13 Guy Even, Orr Fischer, Pierre Fraigniaud, Tzlil Gonen, Reut Levi, Moti Medina, Pedro Montealegre, Dennis Olivetti, Rotem Oshman, Ivan Rapaport, and Ioan Todinca. Three Notes on Distributed Property Testing. In *DISC*, 2017.

14 M. R. Fellows, A. Kulik, F. Rosamond, and H. Shachnai. Parameterized approximation via fidelity preserving transformations. *J. of Computer and System Sciences*, 2018.

15 Manuela Fischer. Improved Deterministic Distributed Matching via Rounding. In *DISC*, 2017.

16 Silvio Frischknecht, Stephan Holzer, and Roger Wattenhofer. Networks Cannot Compute Their Diameter in Sublinear Time. In *SODA*, 2012.

17 Mohsen Ghaffari and Bernhard Haeupler. Distributed Algorithms for Planar Networks II: Low-Congestion Shortcuts, MST, and Min-Cut. In *SODA*, 2016.

18 Mohsen Ghaffari, Fabian Kuhn, and Yannic Maus. On the Complexity of Local Distributed Graph Problems. In *STOC 2017*, 2017.

19 Bernhard Haeupler, Jason Li, and Goran Zuzic. Minor Excluded Network Families Admit Fast Distributed Algorithms. In *PODC*, 2018.

20 D. G. Harris. Distributed approximation algorithms for maximum matching in graphs and hypergraphs. *ArXiv e-prints*, abs/1807.07645, 2018. `arXiv:1807.07645`.

21 Janne H. Korhonen and Joel Rybicki. Deterministic Subgraph Detection in Broadcast CONGEST. In *OPODIS*, 2017.

22 Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. Local Computation: Lower and Upper Bounds. *J. ACM*, 2016.

23 Christoph Lenzen and Boaz Patt-Shamir. Fast Routing Table Construction Using Small Messages: Extended Abstract. In *STOC*, 2013.

24 Christoph Lenzen and Roger Wattenhofer. Leveraging Linial's locality limit. In *DISC*, 2008.

25 Zvi Lotker, Boaz Patt-Shamir, and Seth Pettie. Improved Distributed Approximate Matching. *J. ACM*, 2015.

26 Zvi Lotker, Boaz Patt-Shamir, and Adi Rosén. Distributed Approximate Matching. *SIAM J. Comput.*, 2009.

27 Dániel Marx. Parameterized complexity and approximation algorithms. *The Comp. J.*, 2008.

28 Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. Distributed verification and hardness of distributed approximation. In *STOC*, 2011.

Message Reduction in the LOCAL Model Is a Free Lunch

Shimon Bitton
Technion – Israel Institute of Technology, Haifa, Israel
sbitton@technion.ac.il

Yuval Emek
Technion – Israel Institute of Technology, Haifa, Israel
yemek@technion.ac.il

Taisuke Izumi
Nagoya Institute of Technology, Japan
t-izumi@nitech.ac.jp

Shay Kutten
Technion – Israel Institute of Technology, Haifa, Israel
kutten@ie.technion.ac.il

Abstract

A new *spanner* construction algorithm is presented, working under the *LOCAL* model with unique edge IDs. Given an n-node communication graph, a spanner with a constant stretch and $O(n^{1+\varepsilon})$ edges (for an arbitrarily small constant $\varepsilon > 0$) is constructed in a constant number of rounds sending $O(n^{1+\varepsilon})$ messages whp. Consequently, we conclude that every t-round LOCAL algorithm can be transformed into an $O(t)$-round LOCAL algorithm that sends $O(t \cdot n^{1+\varepsilon})$ messages whp. This improves upon all previous message-reduction schemes for LOCAL algorithms that incur a $\log^{\Omega(1)} n$ blow-up of the round complexity.

2012 ACM Subject Classification Theory of computation → Sparsification and spanners; Theory of computation → Distributed algorithms

Keywords and phrases distributed graph algorithms, spanner, LOCAL model, message complexity

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.7

Funding *Yuval Emek*: This work has been funded in part by an Israeli Ministry of Science and Technology grant number 3-13565.
Taisuke Izumi: This work was supported by JST SICORP Grant Number JPMJSC1606 and JSPS KAKENHI Grant Number JP19K11824, Japan.
Shay Kutten: This work has been funded in part by an Israeli Ministry of Science and Technology grant number 3-13565.

1 Introduction

What is the minimum number of messages that must be sent by *distributed graph algorithm* for solving a certain task? Is there a tradeoff between the message and time complexities of such algorithms? How do the message complexity bounds depend on the exact model assumptions? These questions are among the most fundamental ones in distributed computing with a vast body of literature dedicated to their resolution.

A graph theoretic concept that plays a key role in this regard is that of *spanners*. Introduced by Peleg and Ullman [33] (see also [32]), an α-*spanner*, or a spanner with *stretch* bound α, of a connected graph $G = (V, E)$ is a (spanning) subgraph $H = (V, S)$ of G where

© Shimon Bitton, Yuval Emek, Taisuke Izumi, and Shay Kutten;
licensed under Creative Commons License CC-BY
33rd International Symposium on Distributed Computing (DISC 2019).
Editor: Jukka Suomela; Article No. 7; pp. 7:1–7:15

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

the distance between any two vertices is at most α times their distance in G.[1] More general spanners, called (α, β)-*spanners*, are also considered, where the spanner distance between any two nodes is at most α times their distance in G plus an additive β-term ([17]).

Sparse low stretch spanners are known to provide the means to save on message complexity in the *LOCAL* model [27,31] without a significant increase in the round complexity. This can be done via the following classic simulation technique: Given an n-node communication graph $G = (V, E)$ and a LOCAL algorithm $\mathcal{A}$ whose run $\mathcal{A}(G)$ on G takes t rounds, (1) construct an α-spanner $H = (V, S)$ of G; and (2) simulate each communication round of $\mathcal{A}(G)$ by α communication rounds in H so that a message sent over the edge $(u, v) \in E$ under $\mathcal{A}(G)$ is now sent over a (u, v)-path of length at most α in H. The crux of this approach is that the simulating algorithm executed in stage (2) runs for αt rounds and sends at most $2\alpha t \cdot |S|$ messages. Therefore, if α and $|S|$ are "small", then the simulating algorithm incurs "good" round and message bounds. In particular, the performance of the simulating algorithm does not depend on the number $|E|$ of edges in the underlying graph G.

What about the performance of the spanner construction in the "preprocessing" stage (1) though? A common thread among distributed spanner construction algorithms is that they all send $\Omega(|E|)$ messages when running on graph $G = (V, E)$. Consequently, accounting for the messages sent during this preprocessing stage, the overall message complexity of the aforementioned simulation technique includes a seemingly inherent $\Omega(|E|)$ term. The following research question that lies at the heart of distributed message reduction schemes is therefore left open.

▶ **Question 1.** *Given a LOCAL algorithm $\mathcal{A}$ whose run $\mathcal{A}(G)$ on G takes t rounds, is it possible to simulate $\mathcal{A}(G)$ in $O(t)$ rounds while sending only $O(n^{1+\varepsilon})$ messages for an arbitrarily small constant $\varepsilon > 0$, irrespective of the number $|E|$ of edges in G?*

This question would be resolved on the affirmative if one could design a LOCAL algorithm that constructs an α-spanner $H = (V, S)$ of G with stretch $\alpha = O(1)$ and $|S| = O(n^{1+\varepsilon})$ edges in $O(1)$ rounds sending $O(n^{1+\varepsilon})$ messages. Despite the vast amount of literature on distributed spanner construction algorithms [5, 9–11, 14, 16, 18, 35], it is still unclear if such a LOCAL spanner construction algorithm exists.

Some progress towards the positive resolution of Question 1 has been obtained by Censor-Hillel et al. [8] and Haeupler [22] who introduced techniques for simulating LOCAL algorithms by *gossip* processes. Using this approach, one can transform any t-round LOCAL algorithm into a LOCAL algorithm that runs in $O(t \log n + \log^2 n)$ rounds while sending n messages per round [22]. This transformation provides a dramatic message complexity improvement if one is willing to accept algorithms that run for $\log^{O(1)} n$ many rounds, e.g., if the the bound t on the round complexity of the original algorithm is already in the $\log^{O(1)} n$ range. However, if $t = \log^{o(1)} n$, then the gossip based message reduction scheme of [8, 22] significantly increases the round complexity and this increase seems to be inherent to that technique.

1.1 Definitions and Results

Throughout, we consider a communication network represented by a connected unweighted undirected graph $G = (V, E)$ and denote $n = |V|$. The nodes of G participate in a distributed algorithm under the (fully synchronous) *LOCAL* model [27,31] with the following two model

[1] An equivalent definition requires that it admits a path of length at most α between any two nodes adjacent in G.

assumptions: (i) the nodes know an $O(1)$-approximate upper bound on $\log n$ (equivalently, a poly(n)-approximate upper bound of n) at all times; and (ii) the graph admits *unique edge IDs* so that the ID of an edge is known to both its endpoints at all times.[2] Other than that, the nodes have no a priori knowledge of G's topology. Our main technical contribution is a new algorithm for constructing sparse spanners, called `Sampler`, whose guarantees are cast in the following Theorem.

▶ **Theorem 2.** *Fix integer parameters $1 \leq k \leq \log \log n$ and $0 \leq h \leq \log n$. Algorithm* `Sampler` *constructs an edge set $S \subseteq E$ of size $|S| \leq \tilde{O}(n^{1+1/(2^{k+1}-1)})$ such that $H = (V, S)$ is an $O(3^k)$-spanner of G whp.*[3][4] *The round complexity of* `Sampler` *is $O(3^k h)$ and its message complexity is $\tilde{O}(n^{1+1/(2^{k+1}-1)+(1/h)})$ whp.*

By setting the parameters k and h so that $1/(2^{k+1} - 1) = 1/h = \varepsilon/2$ for an arbitrarily small constant $\varepsilon > 0$ and utilizing the aforementioned spanner based simulation technique, we obtain a message-reduction scheme that transforms any LOCAL algorithm $\mathcal{A}$ whose run on G takes t rounds into a (randomized) LOCAL algorithm that runs in $O(t)$ rounds and sends $\tilde{O}(tn^{1+\varepsilon})$ messages whp. This resolves Question 1 on the affirmative provided that one is willing to tolerate a $1/\text{poly}(n)$ error probability. In fact, we can improve the message reduction scheme even further via the following two-stage process: first, use the α-spanner $H = (V, S)$ constructed by `Sampler` to simulate the run on G of some off-the-shelf LOCAL algorithm that constructs an α'-spanner $H' = (V, S')$ with a better tradeoff between α' and $|S'|$; then, use H' to simulate the run of $\mathcal{A}$ on G. In Section 6, we show that with the right choice of parameters, this two-stage process leads to the following theorem.

▶ **Theorem 3.** *Every distributed task solvable by a t-round LOCAL algorithm can be solved with any one of the following pairs of time and message complexities:*

- *$\tilde{O}(tn^{1+2/(2^{\gamma+1}-1)})$ message complexity and $O(3^\gamma t + 6^\gamma)$ round complexity for any $1 \leq \gamma \leq \log \log n$,*
- *$\tilde{O}(t^2 n^{1+O(1/\log t)})$ message complexity and $O(t)$ round complexity.*

1.2 Related Work and Discussion

Model Assumptions

The current paper considers the fully synchronous message passing LOCAL model [27, 31] that ignores the message size and focuses only on locality considerations. This model has been studied extensively (at least) since the seminal paper of Linial [27], with special attention to the question of what can be computed efficiently, including some recent interesting developments, see, e.g., the survey in [20, Section 1]. The more restrictive *CONGEST* model [31], where message size is bounded (typically to $O(\log n)$ bits), has also been extensively studied.

Many variants of the LOCAL model have been addressed over the years, distinguished from each other by the exact model assumptions, the most common such assumptions being unique node IDs and knowledge of n. Another important distinction addresses the exact knowledge held by any node v regarding its incident edges when the execution commences.

[2] Alternatively, the algorithm can run under the rather common KT_1 model variant [3], where the nodes are associated with unique IDs and each node knows the ID of the other endpoint of each one of its incident edges; see the discussion in Section 1.2.

[3] We say that an event occurs *with high probability*, abbreviated by *whp*, if the probability that it does not occur is at most n^{-c} for an arbitrarily large constant c.

[4] The asymptotic notation $\tilde{O}(\cdot)$ may hide $\log^{O(1)} n$ factors.

Two common choices in this regard are the KT_0 variant, where v knows only its own degree, and the KT_1 variant, where v knows the ID of e's other endpoint for each incident edge e [3]. The authors of [3] advocate KT_1, arguing that it is the more natural among the two model variants, but papers have been published about each of them.

In the current paper, it is assumed that each *edge* $(u, v) \in E$ is equipped with a unique ID, known to both u and v. In general, this assumption lies (strictly) between the KT_0 and KT_1 model variants. Note that the unique edge IDs assumption is no longer weaker than the KT_1 assumption when the communication graph admits *parallel* edges. However, our algorithm and analysis apply also to such graphs (assuming that $|E| \leq n^{O(1)}$) under either of the two assumptions.

Message Complexity $o(|E|)$

As discussed in Section 1.1, the main conceptual contribution of this paper is that on graphs with $m = |E| \gg n$ edges, many distributed tasks can now be solved by sending $o(m)$ messages while keeping the round complexity unharmed. The challenge of reducing the message complexity below $O(m)$ has already received significant attention. In particular, it has been proved in [25] that under the CONGEST KT_0 model, intensively studied *global* tasks, namely, distributed tasks that require $\Omega(D)$ rounds, where D is the graph's diameter (e.g., broadcasting, leader election, etc.), cannot be solved unless $\Omega(m)$ messages are sent (in the worst case).

This is no longer true under more relaxed models. For example, under the LOCAL KT_1 model, DFS and leader election can be solved by sending $O(n)$ and $O(n \log n)$ messages, respectively [24]. This implies similar savings in the number of messages required for most global tasks (trivially, by collecting all the information to the leader). Under the CONGEST KT_1 model, it has been recently proved that a *minimum spanning tree* can be constructed, sending $o(m)$ messages [19, 21, 23, 29].

Restricted graph classes have also been addressed in this regard. In particular, the authors of [26] proved that under the CONGEST KT_0 model, the message complexity of leader election is $O(\sqrt{n} \log^{\frac{3}{2}} n)$ whp in the complete graph and more generally, $O(\tau \sqrt{n} \log^{\frac{3}{2}} n)$ whp in graphs with mixing time $\tau(G)$.

Spanners

Graph spanners have been extensively studied and papers dealing with this fundamental graph theoretic object are too numerous to cite. Beyond the role that sparse spanners play in reducing the message complexity of distributed (particularly LOCAL) algorithms as discussed in Section 1.1, spanners have many applications in various different fields, some of the more relevant ones include synchronization [1, 33], routing [2, 34], and distance oracles [36].

Many existing distributed spanner algorithms have a node collect the topology of the graph up to a distance of some r from itself [9, 12] or employ more sophisticated bounded diameter graph decomposition techniques involving the node's neighborhood [13, 16, 18, 30] such as the techniques presented in [6, 15, 28]. This approach typically requires sending messages over every edge at distance at most r from some subset of the nodes which leads to a large number of messages. Another approach to constructing sparse spanners in a distributed manner is to recursively grow local clusters [4, 5, 7, 10, 35]. Although this approach does not require the (explicit) exploration of multi-hop neighborhoods, the existing algorithms operating this way also admit large message complexity because too many nodes have to explore their 1-hop neighborhoods. Our algorithm `Sampler` is inspired by the algorithm of [5] and adheres to the latter approach, but it is designed in a way that drastically reduces the message complexity – see Section 1.3.

1.3 Techniques Overview

Algorithm `Sampler` employs *hierarchical node sampling*, where a sampled node u in level j of the hierarchy forms the *center* of a cluster that includes (some of) its non-sampled neighbors v. An edge connecting u and v is added to the spanner. The clusters are then contracted into the nodes of the next level $j+1$. Also added to the spanner are all incident edges of every non-center node that has no adjacent center.

This hierarchical node sampling is used also in the distributed spanner construction of Baswana and Sen [5] and similar recursive clustering techniques were used in other papers as well (see Section 1.2). Common to all these papers is that the centers in each level communicate directly with *all* their neighbors to facilitate the cluster forming task.[5] As this leads inherently to $\Omega(|E|)$ messages, we are forced to follow a different approach, also based on (a different) sampling process.

The first thing to notice in this regard is that it is enough for a non-center node v to find a *single* center u in its neighborhood, so perhaps there is no need for the center nodes to announce their status to *all* their neighbors? Indeed, we invoke an *edge sampling* process in the non-center nodes v that identifies a subset of v's incident edges over which *query* messages are sent. Our analysis shows that (1) the number of query messages is small whp; (2) if v does have an adjacent center, then the edge sampling process finds such a center whp; and (3) if v does not have an adjacent center, then v's degree is small; in this case, all its incident edges are queried and join the spanner whp.

However, this edge sampling idea by itself does not suffice: Note that the graph (in some level of the hierarchical construction) is constructed via cluster contraction (into a single node) in lower levels of the hierarchy. Hence this graph typically exhibits edge multiplicities (even if the original communication graph is simple). This means that some neighbors w of v may have many more (parallel) (w, v)-edges than others. Informally, this can bias the probabilities for finding additional neighbors in the edge sampling process. The key idea in resolving this issue is to run the edge sampling process (in each level) in a carefully designed *iterative* fashion. Intuitively, the first iterations in each level "peels off" the neighbors to which a large fraction of v's incident edges lead. This increases the probability of finding one of the rest of the neighbors in later iterations. Note that once v found a neighbor u, node v can identify all v's edges leading to u (and so "peel them off" from the next iterations), by having u report to v the IDs of all the edges touching u.

1.4 Paper Organization

The rest of the paper is organized as follows. Following some preliminary definitions provided in Section 2, `Sampler` is presented in Section 3 and analyzed in Section 4. For clarity of the exposition, we first present `Sampler` as a centralized algorithm and then, in Section 5, explain how it can be implemented under the LOCAL model with the round and message complexities promised in Theorem 2. Section 6 describes how algorithm `Sampler` is used to obtain the message-reduction schemes we mentioned in the introduction. Finally, we conclude the paper in Section 7.

[5] Some of these papers consider weighted communication graphs and as such, have to deal with other issues that are spared in the current paper.

2 Preliminaries

Consider some graph $G = (V, E)$. When convenient, we may denote the node set V and edge set E by $V(G)$ and $E(G)$, respectively. Unless stated otherwise, the graphs considered throughout this paper are undirected and not necessarily simple, namely, the edge set E may include edge multiplicities (a.k.a. *parallel* edges). Given disjoint node subsets $U, U' \subseteq V$, let $E(U)$ denote the subset of edges with (exactly) one endpoint in U and let $E(U, U') = E(U) \cap E(U')$ denote the subset of edges with one endpoint in U and the other in U'. If $U = \{u\}$ and $U' = \{u'\}$ are singletons, then we may write $E(u)$ and $E(u, u')$ instead of $E(\{u\})$ and $E(\{u\}, \{u'\})$, respectively (notice that $E(u, u')$ may contain multiple edges when G is not a simple graph).

Let $\mathcal{C} = \{C_1, \ldots, C_\ell\}$ be a collection of non-empty pairwise disjoint node subsets referred to as *clusters* of G.[6] The *cluster graph* (cf. [31]) induced by $\mathcal{C}$ on G, denoted by $G(\mathcal{C})$, is the undirected graph whose nodes are identified with the clusters in $\mathcal{C}$, and the edges connecting nodes C_i and C_j, $1 \leq i \neq j \leq \ell$, correspond to the edges crossing between clusters C_i and C_j in G, that is, the edges in $E(C_i, C_j)$. Observe that $G(\mathcal{C})$ may include edge multiplicities even if G is a simple graph. We denote by $\mathrm{Ind}_G(C)$ the subgraph of G induced by C. When convenient, the term "cluster C applies also to $\mathrm{Ind}_G(C)$[7] For $u, v, \in V$, the distance between u and v in G is denoted by $\mathrm{dist}_G(u, v)$.

As stated in the introduction, the assumption about global parameters is that each node knows an $O(1)$-approximate upper bound on $\log n$. For the sake of simplicity, we treat the algorithm as if each node knows the exact value of $\log n$, however, this is not essential.

3 Constructing an $O(3^k)$-Spanner

In the following argument, let $\delta = 1/(2^{k+1} - 1)$ and $\epsilon = 1/h$ for short. Denote the simple graph input to the algorithm by $G_0 = (V_0, E_0)$. Algorithm `Sampler` (see Pseudocode 1) generates a sequence $G_1, \ldots, G_k$ of graphs, where $G_j = (V_j, E_j)$. Let n_j and m_j be the numbers of nodes and edges in G_j respectively, $N_j(v)$ be the set of neighbors in G_j of node $v \in V_j$, and $E_j(v, u)$ be the set of edges connecting v to u in G_j. The process is executed in an iterative fashion, where in each iteration $j = 0, \ldots, k$, the algorithm constructs a collection $\mathcal{C} \subset 2^{V_j}$ of pairwise disjoint clusters and an edge set $F \subseteq E_j$ that is added to the spanner edge set S (as an exception, in the final iteration of $j = k$, only F is constructed but the cluster collection $\mathcal{C}$ is not created). The graph G_{j+1} is then defined to be the cluster graph $G_j(\mathcal{C})$ induced by $\mathcal{C}$ on G_j. The construction of $\mathcal{C}$ and F is handled by procedure $\texttt{Cluster}_j$ that is described soon. To avoid confusion, in what follows, we fix $n = n_0 = |V_0|$.

Procedure $\texttt{Cluster}_j$

On input graph G_j $(0 \leq j \leq k)$, this procedure constructs the cluster collection $\mathcal{C} \subset 2^{V_j}$ and edge set $F = \cup_{v \in V_j} F_v$ $(F_v \subseteq E_j)$ to be added to the spanner edges. The procedure (see Pseudocode 2) consists of two steps. In the first step, each node v tries to identify $\min\{c \exp_n(2^j \delta) \log n, |N_j(v)|\}$ neighbors by an iterative random-edge sampling process[8],

[6] Notice that the union of the clusters is not required to be the whole node set V.
[7] Each cluster here will be a connected component.
[8] For ease of writing long exponents, define $\exp_a(x) = a^x$.

Algorithm 1 `Sampler`.

1: $n \leftarrow |V_0|$; $\delta \leftarrow 1/(2^{k+1}-1)$; $S \leftarrow \emptyset$
2: **for** $j = 0, \dots, k$ **do**
3: $\quad \langle \mathcal{C}, F \rangle \leftarrow \mathtt{Cluster}_j$
4: $\quad S \leftarrow S \cup F$
5: $\quad$ **if** $j < k$ **then**
6: $\quad\quad G_{j+1} \leftarrow G_j(\mathcal{C})$
7: return S

where c is a sufficiently large constant to guarantee high success probability of the algorithm. For this process, v maintains a set $X_v \subseteq E_j(v)$ of the edges that have not been explored yet. Initially, X_v is set to $X_v = E_j(v)$; the content of X_v is then gradually eliminated by running $2/\epsilon = 2h$ *trials*. In every trial, each node $v \in V_j$ chooses $c^2 \exp_n(2^j\delta + \epsilon) \log^3 n$ edges from X_v independently and uniformly at random (possibly choosing the same edge twice or more). Each chosen edge is said to be a *query* edge. For a neighbor $u \in N_j(v)$ such that $E_j(v, u)$ contains a query edge, we say that u is *queried* by v. For each queried node $u \in N_j(v)$, v adds one arbitrary query edge $e \in E_j(u, v)$ to the edge set F_v, and eliminates all the edges in $E_j(u, v)$ from X_v (Figure 1 (a)-(c)). Then the procedure advances to the next trial unless $|F_v| \geq c \exp_n(2^j\delta) \log n$ holds or X_v is emptied. Let $\hat{N}_j(v) \subseteq N_j(v)$ be the set of the nodes queried by v after finishing $2h$ trials. A node $v \in V_j$ is called *light* if $\hat{N}_j(v) = N_j(v)$ holds, or called *heavy* if $|N_j(v)| > |\hat{N}_j(v)| \geq c \exp_n(2^j\delta) \log n$. It is proved later that every node becomes either light or heavy whp.

In the second step, the algorithm creates (only if $j < k$) the vertex set V_{j+1} by clustering the nodes in V_j. The algorithm marks each node $v \in V_j$ as a *center* w.p. $p_j = \exp_n(-2^j\delta)$. Each node v having a center u contained in $\hat{N}_j(v)$ is merged into u (if two or more centers are contained, an arbitrary one is chosen). A merged cluster corresponds to a node in G_{j+1} (Figure 1 (d)-(f)). As a result, letting $\mathcal{C} \subset 2^{V_j}$ be the clusters inducing the cluster graph G_{j+1}, each cluster $C = C(u) \in \mathcal{C}$ contains exactly one center $u \in V_j$ and some subset of $N_j(u)$. The node which is not merged into any center is said to be an *unclustered* node. Due to some technical reason, all the nodes in G_k are defined to be unclustered. It is shown that every heavy node is merged into some center whp., and that every node in G_k is light (proved later). Thus, every unclustered node is light.

4 Analysis

Throughout this section, we refine the definition of terminology "whp." to claim that the probabilistic event considered in the context holds with probability $1 - 1/n^{\Theta(c)}$ for parameter c defined in the algorithm. With a small abuse of probabilistic arguments, we treat those events as if they necessarily occur (with probability one). Since we only handle a polynomially-bounded number of probabilistic events in the proof, the standard union-bound argument ensures that any consequence of the analysis also holds whp. for a sufficiently large c. We begin the analysis by bounding the number of nodes in graph G_j. Denote $\hat{p}_j = \prod_{0 \leq i \leq j} p_i$. It is easy to check that $\hat{p}_{j-1} = \exp_n(-(2^j - 1)\delta)$ for $1 \leq j \leq k$.

▶ **Lemma 4.** *Graph G_j satisfies $n\hat{p}_{j-1}/2 \leq n_j \leq 3n\hat{p}_{j-1}/2$ whp. for $1 \leq j \leq k$.*

Algorithm 2 `Cluster`$_j$.

1: **for all** $v \in V_j$ **do**
2: $F'_v \leftarrow \emptyset$; $F_v \leftarrow \emptyset$; $X_v \leftarrow E_j(v)$; $i \leftarrow 1$
3: /∗ First Step ∗/
4: **while** $(i \leq 2h) \wedge (|F'_v| < c\exp_n(2^j\delta)\log n) \wedge (|X_v| \neq \emptyset)$ **do** ▷ run (at most) $2h$ trials
5: **for** $x = 1$ to $c^2\exp_n(2^j\delta + \epsilon)\log^3 n$ **do**
6: add an edge e selected uniformly at random from X_v to F'_v
7: **while** $(F'_v \setminus F_v) \neq \emptyset$ **do** ▷ $F'_v \setminus F_v$ is the set of edges newly added in the current trial
8: Pick an arbitrary edge $e = (v, u) \in F'_v \setminus F_v$
9: Remove all the edges incident to u from X_v
10: Remove all the edges incident to u other than e from F'_v
11: $F_v \leftarrow F_v \cup \{e\}$
12: $i \leftarrow i + 1$
13: $F \leftarrow \cup_{v \in V_j} F_v$
14: /∗ Second Step ∗/
15: **if** $j < k$ **then**
16: **for all** $v \in V_j$ **do**
17: mark v as a center and create $C(v) = \{v\}$ w.p. $\exp_n(-2^j\delta)$
18: **for all** non-center $v \in V_j$ **do**
19: **if** $\exists(v, u) \in F : u$ is a center **then**
20: $C(u) \leftarrow C(u) \cup \{v\}$
21: return $\langle\{C(u) \mid u \text{ is a center}\}, F\rangle$

Proof. The value n_j follows the binomial distribution of n_{j-1} trials and success probability p_{j-1}. Applying Chernoff bound (under conditioning n_{j-1}), the inequality below holds for all $1 \leq j \leq k$ whp.[9]

$$\left(1 - \sqrt{\frac{c\log n}{n_{j-1}p_{j-1}}}\right) n_{j-1}p_{j-1} \leq n_j \leq \left(1 + \sqrt{\frac{c\log n}{n_{j-1}p_{j-1}}}\right) n_{j-1}p_{j-1}.$$

By the definition, n_j and p_j are both non-increasing with respect to j. Hence we have

$$\left(1 - \sqrt{\frac{c\log n}{n_{j-1}p_{j-1}}}\right)^{j-1} n\hat{p}_{j-1} \leq n_j \leq \left(1 + \sqrt{\frac{c\log n}{n_{j-1}p_{j-1}}}\right)^{j-1} n\hat{p}_{j-1}.$$

We prove the lemma inductively (on j) following this inequality. For $j = 1$, $n_{j-1} = n$ and thus $|\sqrt{c\log n/n_{j-1}p_{j-1}}| \leq 1/2$ holds. From the inequality above, we obtain $n_0p_0/2 \leq n_1 \leq 3n_0p_0/2$. Suppose $n\hat{p}_{j-2}/2 \leq n_{j-1} \leq 3n\hat{p}_{j-1}/2$. Since $n\hat{p}_{j-1} = \exp_n(1 - (2^j - 1)\delta) \geq n^{1/2}$ holds for $j \leq k$, we have $\sqrt{c\log n/(n_{j-1}p_{j-1})} \leq c\log n/n^{1/2} \ll 1/2(j-1)$ for sufficiently large n. Applying the approximation of $(1+x)^y \approx 1 + xy$ for $|x| \ll 1$ to the inequality, we obtain the lemma. ◀

[9] Chernoff bound for the binomial distribution X of m trials and success probability p is $\mathbb{P}[|X - mp| \geq \alpha mp] \leq 2e^{-\frac{\alpha^2 mp}{3}}$.

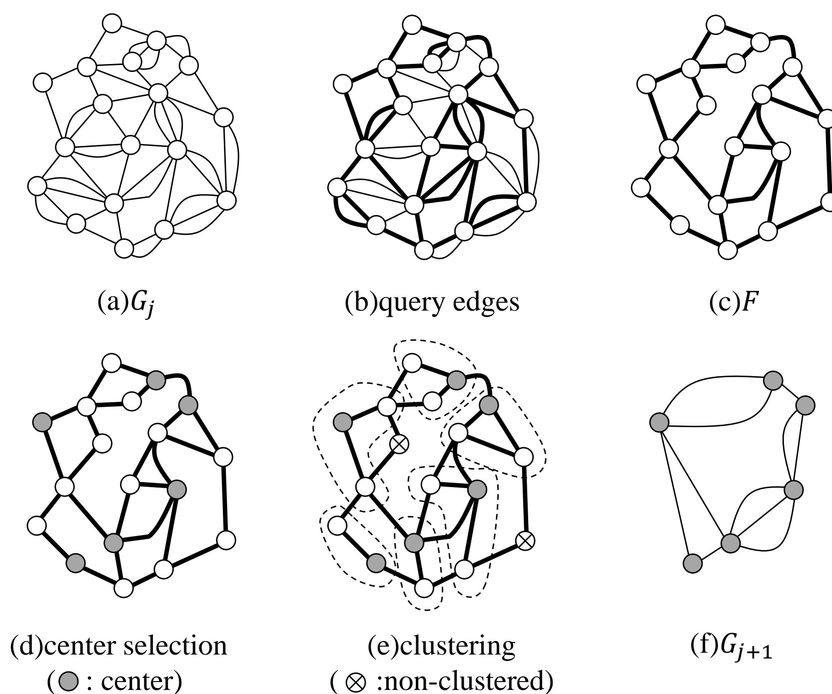


Figure 1 Procedure $\texttt{Cluster}_j$.

We next prove the facts mentioned in the explanation of the procedure $\texttt{Cluster}_j$.

▶ **Lemma 5.** *For any $0 \le j \le k-1$, any heavy node $v \in V_j$ contains at least one center in $\hat{N}_j(v)$ whp.*

Proof. The probability that no center is contained in $\hat{N}_j$ is $(1-p_j)^{|\hat{N}_j(v)|}$. Since $|\hat{N}_j(v)| \ge c\exp_n(2^j\delta)\log n = c\log n/p_j$ holds for any heavy node v, the probability is at most $1/n^c$. ◀

▶ **Lemma 6.** *For any $0 \le j \le k$, any node $v \in V_j$ becomes light or heavy whp. Furthermore, any node $v \in V_k$ becomes light whp.*

Proof. Let $\alpha = (3c\exp_n(2^j\delta)\log^2 n)/h$ for short. For $W \subseteq N_j(v)$, define $E_j(v,W) = \bigcup_{u \in W} E_j(v,u)$. Let $N^i_j(v) \subseteq N_j(v)$ be the set of the nodes not queried by v at the beginning of the i-th trial, and m_i be the numbers of edges in X_v at the beginning of the i-th trial. For any node $u \in N^i_j(v)$, the value of $|E_j(v,u)|$ is called the *volume* of u. Similarly, for any $X \subseteq N^i_j(v)$, we call the value of $|E_j(v,X)|$ the volume of X. Divide $N^i_j(v)$ into $2/\epsilon = 2h$ classes: A node $u \in N^i_j(v)$ belongs to the x-th class $\mathcal{K}^i_x(v)$ if $|E_j(v,u)| \in (\exp_n(x\epsilon), \exp_n((x+1)\epsilon)]$ holds $(0 \le x \le 2h-1)$. Let $\mathcal{K}^i(v)$ be the maximum-volume class of all at the beginning of the i-th trial. Since the volume of $\mathcal{K}^i(v)$ is at least $m_i/2h$, there exists at least one node $u \in \mathcal{K}^i(v)$ satisfying $|E_j(v,u)| \ge m_i/(2h|\mathcal{K}^i(v)|)$. By the definition of class $\mathcal{K}^i(v)$, it implies that the volume of any node $u \in \mathcal{K}^i(v)$ is at least $m_i/(2h|\mathcal{K}^i(v)|n^\epsilon)$.

Let β be the non-negative integer satisfying $(\beta-1)\alpha \le |K^i(v)| \le \beta\alpha$. Then we consider an arbitrary partition of $\mathcal{K}^i(v)$ into $q = \lfloor |\mathcal{K}^i(v)|/\beta \rfloor$ groups $K_1, K_2, \ldots, K_q$ of size β. Note that βq is not necessarily equal to $|\mathcal{K}^i(v)|$, but the residuals are omitted. Since any node in $\mathcal{K}^i(v)$ has a volume at least $m_i/(2h|\mathcal{K}^i(v)|n^\epsilon)$, the volume of K_ℓ $(1 \le \ell \le q)$ is at least

$\beta m_i/(2h|\mathcal{K}^i(v)|n^\epsilon)$. Thus the probability that a query edge is sampled from $E_j(v, K_\ell)$ is at least $\beta/(2h|\mathcal{K}^i(v)|n^\epsilon) \geq 1/(2h\alpha n^\epsilon)$. Letting Z_ℓ be the number of query edges in $E_j(v, K_\ell)$ created at the i-th trial, for any $1 \leq \ell \leq q$, we have

$$\mathbb{P}[Z_\ell = 0] \leq \left(1 - \frac{1}{2h\alpha n^\epsilon}\right)^{c^2 \exp_n(2^j\delta+\epsilon)\log^3 n} \leq \exp_e\left(-\frac{c^2 \exp_n(2^j\delta)\log^3 n}{2h\alpha}\right) \leq e^{-c\log n/6}.$$

Thus, in every trial, at least one node in each group K_ℓ is queried by v whp. If $|\mathcal{K}^i(v)| \leq \alpha$ holds, $\beta = 1$ holds and thus each group consists of a single node in $\mathcal{K}^i(v)$. Thus all nodes in $\mathcal{K}^i(v)$ are queried by v whp. in the i-th trial (note that no node becomes a residual in the case of $\beta = 1$). Otherwise, $q = \lfloor|\mathcal{K}^i(v)|/\beta\rfloor \geq \lfloor(\beta-1)\alpha/\beta\rfloor \geq \alpha/3 = (c\exp_n(2^j\delta)\log^2 n)/h \geq c\exp_n(2^j\delta)\log n$ holds, and thus v queries at least $c\exp_n(2^j\delta)\log n$ nodes in the i-th trial. Consequently, if $|\mathcal{K}^i(v)| \leq \alpha$ holds for all $1 \leq i \leq 2h$, v queries all nodes in $\mathcal{K}^i(v)$ at the i-th trial, that is, queries all nodes in $N_j(v)$ throughout the run of $\texttt{Cluster}_j$. Then v becomes light. If $|\mathcal{K}^i(v)| > \alpha$ holds for some i, v queries at least $c\exp_n(2^j\delta)\log n$ nodes in $\mathcal{K}^i(v)$, which implies v becomes heavy or light.

Finally, let us show that any node $v \in V_k$ is light. Since $n_k \leq 3\exp_n(1-(2^k-1)\delta)/2 = 3\exp_n(2^k\delta)/2 \leq (3c\exp_n(2^k\delta)\log^2 n)/h = \alpha$ holds from Lemma 4, we have $|N_j(v)| \leq n^k \leq \alpha$. Since $\mathcal{K}^i(v)$ is a subset of $N_j(v)$, $|\mathcal{K}^i(v)| \leq \alpha$ holds for all i. By the argument above, then v is light. ◀

The rest of the analysis is divided into two parts: First, in Section 4.1, we analyze the stretch of H, proving that it is at most $\kappa = O(3^k)$. Section 4.2 then establishes an $\tilde{O}(n^{1+\delta})$ upper bound on the number of edges in H.

4.1 Bounding the Stretch

The following lemma is a well-known fact.

▶ **Lemma 7** ([32]). *Let $H = (V, X)$ be any (spanning) subgraph of $G = (V, E)$ and $\mathcal{C}$ be the partition of V such that for any $C \in \mathcal{C}$, $\mathrm{Ind}_H(C)$ has a diameter at most ℓ. If X contains at least one edge in $E(C_i, C_j)$ for any pair $(C_i, C_j) \in \mathcal{C}^2$ such that $E(C_i, C_j)$ is nonempty, H is a $(2\ell+1)$-spanner.*

Let $V'_j \subseteq V_j$ $(1 \leq j \leq k-1)$ be the set of the nodes unclustered in the run of $\texttt{Cluster}_j$, and $V' = \bigcup_{1\leq j\leq k-1} V'_j$. We define $C_j(v) \subseteq V$ as the set of nodes in V which are clustered into $v \in V_j$, and also define $r(v)$ as the value j satisfying $v \in V'_j$ for any $v \in V'$. Let $C(v) = C_{r(v)}(v)$ for short.

▶ **Lemma 8.** *Let $H = (V, S)$ be the (spanning) subgraph output by* `Sampler`*. The diameter of $\mathrm{Ind}_H(C_j(v))$ for any $v \in V_j$ is at most $r = 3^j - 1$.*

Proof. We show that $\mathrm{Ind}_H(C_j(v))$ contains a spanning tree of $\mathrm{Ind}_G(C_j(v))$ with height at most $3^j - 1$. The proof follows the induction on j. For $j = 0$, $\mathrm{Ind}_H(C_j(v)) = \mathrm{Ind}_G(C_j(v))$ is a graph consisting of a single node, and thus its diameter is zero. Suppose as the induction hypothesis that the lemma holds for some j, and consider the case of $j+1$. Since any node $v \in V_{j+1}$ corresponds to a center node $v \in V_j$ and a star-based connection with its neighbors in V_j, $\mathrm{Ind}_H(C_j(v))$ is obviously contains a spanning tree of $\mathrm{Ind}_G(C_j(v))$ with a diameter at most $3(3^j-1)+2 = 3^{j+1}-1$. The lemma follows. ◀

Finally the following theorem is deduced.

▶ **Theorem 9.** *The graph $H = (V, S)$ output by* `Sampler` *is an $O(2 \cdot 3^k - 1)$-spanner of G whp.*

Proof. Since any node in G_k is unclustered, $\mathcal{C} = \{C(v) \mid v \in V'\}$ is a partition of V. If $C(u)$ and $C(v)$ ($u, v \in V'$) are neighboring and $r(u) \le r(v)$ holds, there exists a node $w \in V_{r(u)}$ such that u and w are neighboring in $G_{r(u)}$ and $C_{r(u)}(w) \subseteq C(v)$ holds. Since every unclustered node is light (by Lemmas 5 and 6), $C(u)$ is light. Then at least one edge in $E(C_{r(u)}(u), C_{r(u)}(w))$ is added to S, which implies that at least one edge in $E(C(u), C(v))$ is added to S. Consequently, the edge set S constructed by `Sampler` satisfies the condition of Lemma 7 w.r.t. $\mathcal{C}$. The remaining issue is to bound the diameter of $C(v) \in \mathcal{C}$ for all $v \in V'$, which is shown by Lemma 8. ◀

4.2 Bounding the Number of Edges

Using Lemma 4, we can bound the number of edges in S output by `Sampler`.

▶ **Lemma 10.** *The output edge set S contains $\tilde{O}(n^{1+\delta})$ edges.*

Proof. Each trial of the first step in the run of $\texttt{Cluster}_j$ adds $O(\exp_n(2^j\delta)\log^3 n)$ edges to S per node in V_j, and $n_j = O(\exp_n(1-(2^j-1)\delta))$ holds by Lemma 4. Then the total number of edges added to S in $\texttt{Cluster}_j$ is $h \cdot O(\exp_n(2^j\delta)\log^3 n) \cdot n_j = O(hn^{1+\delta}\log^3 n)$, and thus the size of S is $O(khn^{1+\delta}\log^3 n)$. Since $k, h \le \log n$, we obtain the lemma by omitting all the logarithmic factors. ◀

5 Distributed Implementation

In this section, we explain how the centralized algorithm presented in Section 3 is implemented in a distributed fashion over the (simple) communication graph $G = (V, E)$ with round complexity $O(3^k h)$ and message complexity $\tilde{O}(n^{1+1/(2^{k+1}-1)+(1/h)}) = \tilde{O}(n^{1+\delta+\epsilon})$ whp.

The key observation in this regard is that procedures $\texttt{Cluster}_j$ would have been naturally distributed if the nodes in $V_0, V_1, \dots, V_k$ could have performed local computations and exchanged messages over their incident edges in $G_0, G_1, \dots, G_k$, respectively (recall that graphs $G_1, \dots, G_k$ are virtual, defined only for the sake of the algorithm's presentation). Indeed, the action of marking a node as a center and the action of marking an edge as a query/probe edge are completely local and do not require any communication (in G_j). The action of checking whether a query edge e leads to a center and the action of identifying all the edges parallel to a query/probe edge e are easily implemented under the LOCAL model with unique edge IDs by sending a constant number of messages over edge e in G_j.

So it remains to explain how local actions in graph G_j, $1 \le j \le k$, are simulated in the actual communication graph G. Let $T_j(v)$ be the spanning tree of $\mathrm{Ind}_G(C_j(v))$ shown in the proof of Lemma 8. Once a cluster $C_j(v)$ is formed, no further edge is added to the inside of the cluster. Thus $T_j(v)$ is already contained in $C_j(v)$ at the beginning of the run of $\texttt{Cluster}_j$. In the distributed implementation, the local actions of node v in G_j are simulated by nodes $C_j(v)$ in G via a constant number of broadcast-convergecast sessions over $T_j(v)$ rooted at $v \in V$. (This is made possible by the choice of the LOCAL model with unique edge IDs). This process requires sending $O(1)$ (additional) messages over each edge in $T_j(v)$, and by Lemma 8, it takes at most $O(3^j)$ rounds.

▶ **Theorem 11.** *Algorithm* `Sampler` *has round complexity $O(3^k h)$ and message complexity $\tilde{O}(n^{1+\delta+\epsilon})$ whp.*

Proof. In the first step of $\texttt{Cluster}_j$, each trial takes $O(1)$ rounds in G_j. The second step takes $O(1)$ rounds in G_j. Hence the total running time of $\texttt{Cluster}_j$ takes $O(h(3^j - 1))$ rounds in G. Summing up it over all $0 \leq j \leq k$, the bound on the round complexity is $\sum_{j=0}^{k} O(h(3^j - 1)) = O(3^k h)$.

For the message complexity, the simulation of one round in G_j is implemented with an additive overhead incurred by a constant-number sessions of broadcast and convergecast in each cluster $C(v)$ for $v \in V_j$, which use $O(n)$ messages in total. Each trial of the first step in $\texttt{Cluster}_j$ uses $O(\exp_n(2^j\delta + \epsilon)\log^3 n)$ messages per node in V_j. Thus the total message complexity in $\texttt{Cluster}_j$ is $O(\exp_n(2^j\delta + \epsilon)\log^3 n) \cdot n_j = O(hn^{1+\delta+\epsilon}\log^3 n)$ by Lemma 4. Summing up this over $0 \leq j \leq k$, we can conclude that the message complexity is $O(khn^{1+\delta+\epsilon}\log^3 n) = \tilde{O}(n^{1+\delta+\epsilon})$ (recall $k, h \leq \log n$). ◀

6 Message-Efficient Simulation of Local Algorithms

In this section, we provide a new and versatile message-reduction scheme for LOCAL algorithms based on the new `Sampler` algorithm. The technical ingredients of this scheme consist of a message-efficient t-local broadcast algorithm built on top of constructed spanners, which is commonly used in the past message-reduction schemes [8, 22].

Consider the initial configuration where each node $v \in V$ has a message M_v, and let $B_{G,t}(v) = \{u \mid \mathrm{dist}_G(v, u) \leq t\}$. The task of the t-local broadcast is that each $v \in V$ delivers M_v to all the nodes $u \in B_{G,t}(v)$. In any t-round LOCAL algorithm, the computation at node $v \in V$ relies only on the initial knowledge (i.e., its ID, initial state, and incident edge set) of the nodes in $B_{G,t}(v)$, and thus any t-local broadcast algorithm in the LOCAL model can simulate any t-round LOCAL algorithm. The core of the scheme is the following theorem.

▶ **Lemma 12.** *There exist two t-local broadcast algorithms respectively achieving the following time and message complexities:*

- $\tilde{O}(tn^{1+2/(2^{\gamma+1}-1)})$ *message complexity and* $O(3^\gamma t + 6^\gamma)$ *round complexity for any* $1 \leq \gamma \leq \log\log n$ *and* $t \geq 1$*,*
- $\tilde{O}(t^2 n^{1+1/O(t^{1/\log\log t})\log t)})$ *message complexity and* $O(t)$ *round complexity for* $t \geq 1$.

Proof. Consider the realization of the first algorithm. For any v, all the nodes in $B_{G,t}(v)$ are within αt-hop away from v in any α-spanner. Thus, once we got any α-spanner $H = (V, S)$, the local flooding within distance αt in H trivially implements t-local broadcast. Setting $k = \gamma$ and $h = (2^{\gamma+1} - 1)$ of Theorem 2 implements the spanner satisfying the first condition, where the additive $O(6^\gamma)$ term is the time for spanner construction. For the second algorithm, we utilize the spanner-construction algorithm by Derbel et al. [11] which provides a $(3, O(3^k))$-spanner H with $\tilde{O}(3^k n^{1+1/O(k)})$ edges within $O(3^k)$ rounds for any $k \geq 1$. Consider the algorithm by Derbel et al. with parameter $k = \lceil \log_3 t - \log\log_3 t \rceil$, which results in the $O(t/\log_3 t)$-round algorithm of constructing $(3, O(t))$-spanner with $\tilde{O}(tn^{1+1/O(\log t)})$ edges. We run this algorithm on top of the first simulation scheme with parameter $\gamma = \log_3 \log_3 t$. The simulated algorithm constructs a $(3, O(t))$-spanner H' with $\tilde{O}(tn^{1+1/O(\log t)})$ edges spending $O(3^{\log_3 \log_3 t} \cdot t/\log_3 t + 6^{\log_3 \log_3 t}) = O(t)$ rounds and $\tilde{O}(tn^{1+1/O(\log t)})$ messages. The local flooding within distance $3t + O(t)$ on top of H' implements the t-local broadcast, which takes $O(t)$ rounds and $\tilde{O}(t^2 n^{1+1/O(\log t)})$ messages. The lemma is proved. ◀

As stated above, Theorem 3 is trivially deduced from Lemma 12.

7 Concluding Remarks

In this paper, we present an efficient spanner construction as well as two message-reduction schemes for LOCAL algorithms that preserve the asymptotic time complexity of the original algorithm. The reduced message complexity is close to linear (in n). Is this the best possible in constructing a spanner? Similarly, some open questions still lie on the line of developing efficient message-reduction schemes: (1) While our scheme only sends $\tilde{O}(t^2 n^{1+O(1/\log t)})$ messages for simulating t-round algorithms, it is not clear whether the additive $O(1/\log t)$ term in the exponent can be improved further. Can one have a message-reduction scheme with $\tilde{O}(\mathrm{poly}(t) n^{1+o(1/\log t)})$ message complexity and no overhead in the round complexity? (2) Algorithm `Sampler` inherently relies on randomized techniques for probing the neighbors in G_j using only few messages. Is it possible to obtain a deterministic message-reduction scheme with no degradation of time?

Very recently, the authors received a comment on the first question, which states that utilizing the spanner construction by Elkin and Neiman [16] will improve the message complexity. Unfortunately, due to lack of time, we do not completely check this idea, and thus the current version only states the result based on the algorithm by Derbel et al., but certainly it is a promising approach. If it actually works, the message complexity will be reduced to $\tilde{O}(t^2 n^{1+O(1/t^{1/\log\log t})})$.

Finally, we note that using an $o(m)$ messages spanner construction algorithm that does not increase the time can be useful also for global algorithms in the LOCAL model. It implies that any function can now be computed on the graph in strictly optimal $O(\text{diameter})$ time and $o(m)$ messages (for large enough m).

References

1 Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM (JACM)*, 32(4):804–823, 1985. doi:10.1145/4221.4227.

2 Baruch Awerbuch, Bonnie Berger, Lenore Cowen, and David Peleg. Near-linear time construction of sparse neighborhood covers. *SIAM Journal on Computing (SICOMP)*, 28(1):263–277, 1998. doi:10.1137/S0097539794271898.

3 Baruch Awerbuch, Oded Goldreich, Ronen Vainish, and David Peleg. A trade-off between information and communication in broadcast protocols. *Journal of the ACM (JACM)*, 37(2):238–256, 1990. doi:10.1145/77600.77618.

4 Leonid Barenboim, Michael Elkin, and Cyril Gavoille. A fast network-decomposition algorithm and its applications to constant-time distributed computation. *Theoretical Computer Science*, 751:2–23, 2018. doi:10.1016/j.tcs.2016.07.005.

5 Surender Baswana and Sandeep Sen. A simple and linear time randomized algorithm for computing sparse spanners in weighted graphs. *Random Structures & Algorithms*, 30(4):532–563, 2007. doi:10.1002/rsa.v30:4.

6 Guy E. Blelloch, Anupam Gupta, Ioannis Koutis, Gary L. Miller, Richard Peng, and Kanat Tangwongsan. Nearly-Linear Work Parallel SDD Solvers, Low-Diameter Decomposition, and Low-Stretch Subgraphs. *Theory of Computing Systems*, 55(3):521–554, 2014. doi:10.1007/s00224-013-9444-5.

7 Keren Censor-Hillel and Michal Dory. Distributed Spanner Approximation. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 139–148, 2018. doi:10.1145/3212734.3212758.

8 Keren Censor-Hillel, Bernhard Haeupler, Jonathan Kelner, and Petar Maymounkov. Global Computation in a Poorly Connected World: Fast Rumor Spreading with No Dependence on Conductance. In *Proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing (STOC)*, pages 961–970, 2012. doi:10.1145/2213977.2214064.

9 Bilel Derbel and Cyril Gavoille. Fast Deterministic Distributed Algorithms for Sparse Spanners. *Theoretical Computer Science*, 399(1):83–100, 2008. doi:10.1016/j.tcs.2008.02.019.

10 Bilel Derbel, Cyril Gavoille, David Peleg, and Laurent Viennot. On the Locality of Distributed Sparse Spanner Construction. In *Proceedings of the Twenty-seventh ACM Symposium on Principles of Distributed Computing (PODC)*, pages 273–282, 2008. doi:10.1145/1400751.1400788.

11 Bilel Derbel, Cyril Gavoille, David Peleg, and Laurent Viennot. Local Computation of Nearly Additive Spanners. In *Proceedings of 23rd International Symposium on Distributed Computing (DISC)*, pages 176–190, 2009. doi:10.1007/978-3-642-04355-0_20.

12 Devdatt Dubhashi, Alessandro Mei, Alessandro Panconesi, Jaikumar Radhakrishnan, and Aravind Srinivasan. Fast distributed algorithms for (weakly) connected dominating sets and linear-size skeletons. *Journal of Computer and System Sciences*, 71(4):467–479, 2005. doi:10.1016/j.jcss.2005.04.002.

13 Michael Elkin. Computing Almost Shortest Paths. *ACM Transactions on Algorithms (TALG)*, 1(2):283–323, 2005. doi:10.1145/1103963.1103968.

14 Michael Elkin. A near-optimal distributed fully dynamic algorithm for maintaining sparse spanners. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing (PODC)*, pages 185–194, 2007. doi:10.1145/1281100.1281128.

15 Michael Elkin and Ofer Neiman. Distributed Strong Diameter Network Decomposition: Extended Abstract. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 211–216, 2016. doi:10.1145/2933057.2933094.

16 Michael Elkin and Ofer Neiman. Efficient Algorithms for Constructing Very Sparse Spanners and Emulators. *ACM Transactions on Algorithms (TALG)*, 15(1):4:1–4:29, 2018. doi:10.1145/3274651.

17 Michael Elkin and David Peleg. $(1+\epsilon,\beta)$-Spanner Constructions for General Graphs. *SIAM Journal on Computing*, 33(3):608–631, 2004. doi:10.1137/S0097539701393384.

18 Michael Elkin and Jian Zhang. Efficient algorithms for constructing $(1+\epsilon,\ \beta)$-spanners in the distributed and streaming models. *Distributed Computing*, 18(5):375–385, 2006. doi:10.1007/s00446-005-0147-2.

19 Mohsen Ghaffari and Fabian Kuhn. Distributed MST and Broadcast with Fewer Messages, and Faster Gossiping. In *Proceedings of 32nd International Symposium on Distributed Computing (DISC)*, volume 121, pages 30:1–30:12, 2018. doi:10.4230/LIPIcs.DISC.2018.30.

20 Mohsen Ghaffari, Fabian Kuhn, and Yannic Maus. On the complexity of local distributed graph problems. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 784–797, 2017. doi:10.1145/3055399.3055471.

21 Robert Gmyr and Gopal Pandurangan. Time-Message Trade-Offs in Distributed Algorithms. In *Proceedings of 32nd International Symposium on Distributed Computing, (DISC)*, volume 121, pages 32:1–32:18, 2018. doi:10.4230/LIPIcs.DISC.2018.32.

22 Bernhard Haeupler. Simple, Fast and Deterministic Gossip and Rumor Spreading. *Journal of the ACM (JACM)*, 62(6):47:1–47:18, December 2015. doi:10.1145/2767126.

23 Valerie King, Shay Kutten, and Mikkel Thorup. Construction and impromptu repair of an MST in a distributed network with $o(m)$ communication. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 71–80, 2015. doi:10.1145/2767386.2767405.

24 Ephraim Korach, Shay Kutten, and Shlomo Moran. A modular technique for the design of efficient distributed leader finding algorithms. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):84–101, 1990. doi:10.1145/323596.323611.

25 Shay Kutten, Gopal Pandurangan, David Peleg, Peter Robinson, and Amitabh Trehan. On the Complexity of Universal Leader Election. *Journal of the ACM (JACM)*, 62(1):7:1–7:27, 2015. doi:10.1145/2699440.

26 Shay Kutten, Gopal Pandurangan, David Peleg, Peter Robinson, and Amitabh Trehan. Sublinear bounds for randomized leader election. *Theoretical Computer Science*, 561:134–143, 2015. doi:10.1016/j.tcs.2014.02.009.

27 Nathan Linial. Locality in Distributed Graph Algorithms. *SIAM Journal on Computing (SICOMP)*, 21(1):193–201, 1992. doi:10.1137/0221015.

28 Nathan Linial and Michael Saks. Low diameter graph decompositions. *Combinatorica*, 13(4):441–454, 1993. doi:10.1007/BF01303516.

29 Ali Mashreghi and Valerie King. Broadcast and minimum spanning tree with $o(m)$ messages in the asynchronous CONGEST model. In *Proceedings of 32nd International Symposium on Distributed Computing (DISC)*, pages 37:1–37:17, 2018. doi:10.4230/LIPIcs.DISC.2018.37.

30 Gary L. Miller, Richard Peng, Adrian Vladu, and Shen Chen Xu. Improved Parallel Algorithms for Spanners and Hopsets. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 192–201, 2015. doi:10.1145/2755573.2755574.

31 David Peleg. *Distributed Computing: A Locality-sensitive Approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000. doi:10.1137/1.9780898719772.

32 David Peleg and Alejandro A Schäffer. Graph spanners. *Journal of Graph Theory*, 13(1):99–116, 1989. doi:10.1002/jgt.3190130114.

33 David Peleg and Jeffrey D. Ullman. An Optimal Synchronizer for the Hypercube. *SIAM Journal on Computing (SICOMP)*, 18(4):740–747, 1989. doi:10.1137/0218050.

34 David Peleg and Eli Upfal. A trade-off between space and efficiency for routing tables. *Journal of the ACM (JACM)*, 36(3):510–530, 1989. doi:10.1145/62212.62217.

35 Seth Pettie. Distributed algorithms for ultrasparse spanners and linear size skeletons. *Distributed Computing*, 22(3):147–166, 2010. doi:10.1007/s00446-009-0091-7.

36 Mikkel Thorup and Uri Zwick. Approximate distance oracles. *Journal of the ACM (JACM)*, 52(1):1–24, 2005. doi:10.1145/1044731.1044732.

On the Computational Power of Radio Channels

Mark Braverman
Princeton University, Princeton, NJ, USA
mbraverm@cs.princeton.edu

Gillat Kol
Princeton University, Princeton, NJ, USA
gillat.kol@gmail.com

Rotem Oshman
Tel Aviv University, Israel
roshman@tau.ac.il

Avishay Tal
UC Berkeley, CA, USA
atal@berkeley.edu

Abstract

Radio networks can be a challenging platform for which to develop distributed algorithms, because the network nodes must contend for a shared channel. In some cases, though, the shared medium is an advantage rather than a disadvantage: for example, many radio network algorithms cleverly use the shared channel to approximate the degree of a node, or estimate the contention. In this paper we ask how far the inherent power of a shared radio channel goes, and whether it can efficiently compute "classicaly hard" functions such as Majority, Approximate Sum, and Parity.

Using techniques from circuit complexity, we show that in many cases, the answer is "no". We show that simple radio channels, such as the beeping model or the channel with collision-detection, can be approximated by a low-degree polynomial, which makes them subject to known lower bounds on functions such as Parity and Majority; we obtain round lower bounds of the form $\Omega(n^\delta)$ on these functions, for $\delta \in (0, 1)$. Next, we use the technique of random restrictions, used to prove AC^0 lower bounds, to prove a tight lower bound of $\Omega(1/\epsilon^2)$ on computing a $(1 \pm \epsilon)$-approximation to the sum of the nodes' inputs. Our techniques are general, and apply to many types of radio channels studied in the literature.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Theory of computation → Communication complexity

Keywords and phrases radio channel, lower bounds, approximate majority

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.8

Funding *Mark Braverman*: Research supported in part by NSF Award CCF- 1525342 and the NSF Alan T. Waterman Award, Grant No. 1933331, a Packard Fellowship in Science and Engineering, and the Simons Collaboration on Algorithms and Geometry.
Gillat Kol: Research supported by an Alfred P. Sloan Fellowship, the National Science Foundation CAREER award CCF-1750443, and by the E. Lawrence Keyes Jr. / Emerson Electric Co. Award.
Rotem Oshman: This work was done in part while the author was visiting the Simons Institute for the Theory of Computing. Research supported by the Israeli Centers of Research Excellence (I-CORE) program (Center No.4/11) and by BSF Grant No. 2014256.
Avishay Tal: This work was done in part while the author was visiting the Simons Institute for the Theory of Computing. Partially supported by a Motwani Postdoctoral Fellowship and by NSF grant CCF-1763311.

© Mark Braverman, Gillat Kol, Rotem Oshman, and Avishay Tal;
licensed under Creative Commons License CC-BY
33rd International Symposium on Distributed Computing (DISC 2019).
Editor: Jukka Suomela; Article No. 8; pp. 8:1–8:17

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

In a radio network, nodes communicate over a shared channel, and must contend with each other for access to the channel. This can make some essential distributed tasks challenging, e.g., even broadcasting one piece of information across the network is highly non-trivial (see [34] for a survey). On the other hand, many works have observed that the shared channel also presents some opportunities: in a radio network, nodes can *use* the fact that they contend for the same channel to quickly elect a leader [21], approximate their degree (e.g., using the famous Decay algorithm [7]), approximate local sums, and even approximate the PageRank [29].

In this paper we ask: *what is the computational power of a shared radio channel*? Can it efficiently compute "classical hard functions", like majority or parity? How well can it approximate functions like threshold or sum? We use techniques from circuit complexity to show that the computation power of a shared radio channel is subject to significant limitations, and prove tight lower bounds for several functions.

Part of our motivation comes from the problem of *computing an approximate sum*, which is a useful building block in many radio network algorithms: it can be used to compute an approximate degree or estimate the contention, and also to compute the PageRank and related problems [29] (also see Subsection 1.2). Approximate sum is also used in interactive compression [8, 28], where we require a very good (sub-constant) approximation error. In [29] it is shown that in any beeping network (not just single-hop networks), all nodes can compute a $(1 \pm \epsilon)$ approximation of the sum of their neighbors' inputs, in $O(\text{polylog}(n)/\epsilon^2)$ rounds. Our results show that this quadratic dependence on $1/\epsilon$ is tight, even for the simple single hop topology (although we do not match the polylogarithmic dependence on n).

The model. We consider n wireless nodes with inputs $x_1, \ldots, x_n$, communicating over a shared channel (i.e., a single-hop radio network), with the goal of computing some function $f(x_1, \ldots, x_n)$. Our techniques are quite general, and can handle many types of channels considered in the literature, with or without collision detection. We can also handle an *additive network* (see, e.g., [14]), where the nodes receive the XOR of the bits sent on the channel. Generally, as long as the contents of the channel in every round can be computed by a simple Boolean circuit over the values broadcast by the nodes in that round, the channel will be amenable to our techniques.

For simplicity, in the body of the paper we mostly focus on the single-hop *beeping model*: in each round, every node decides whether to *beep* or not to beep; if at least one node decided to beep, all nodes hear a beep, and otherwise they hear silence. In other words, the beeping channel computes the Boolean OR of the inputs to the channel (i.e., of the individual nodes' decisions whether to beep). The beeping model, and other related models, have received a lot of attention in recent years [2, 36, 3, 24, 19, 21, 30, 10, 23, 18, *etc.*], as they provide an abstraction capturing the simplest possible communication primitive: a detectable burst of "energy". This abstraction is well suited for describing wireless networks. In addition, one of the main motivations for studying the beeping model is due to its connections to signal-driven biological systems [4, 31]: e.g., cells communicating by secreting proteins and other chemical markers that are diffused and sensed by neighboring cells, or fireflies reacting to flashes of light from nearby fireflies.

1.1 Our Results and Techniques

We describe two approaches for bounding the computation power of a radio channel. Again, we focus here on the beeping channel.

1.1.1 Lower Bounds via Polynomial Approximations

Our first approach shows that a protocol for the beeping channel can be *approximated by a low-degree polynomial* over the inputs, where the degree of the polynomial depends on the number of rounds used by the protocol.

▶ **Theorem 1.1.** *If there is a randomized r-round beeping protocol $\mathcal{P}$ that computes $f : \{0,1\}^n \to \{0,1\}$ with error ϵ, then there is a polynomial $g \in \mathbb{F}_2[X_1, \ldots, X_n]$ of degree $O(r^3)$ which agrees with f on all but a 2ϵ-fraction of the inputs in $\{0,1\}^n$.*

Here, $\mathbb{F}_2[X_1, \ldots, X_n]$ denotes the polynomials over $x_1, \ldots, x_n$ with coefficients in $\mathbb{F}_2$.

It is well-known that some functions, such as the parity function, $\text{PARITY}_n(x_1, \ldots, x_n) = \sum_i x_i \bmod 2$, and the majority function, $\text{MAJORITY}_n(x) = [\sum_i x_i \overset{?}{\geq} n/2]$, cannot be approximated by a low-degree polynomial, and this allows us to prove round lower bounds in the beeping model.

▶ **Corollary 1.2.** *Any randomized beeping protocol for* PARITY_n *or* MAJORITY_n *with error* $1/10$ *requires* $\Omega(n^{1/6})$ *rounds.*

Along the way, we show that *any deterministic beeping protocol can be simulated by a constant-depth circuit* with unbounded fan-in AND and OR gates. The size of the circuit corresponds to the number of rounds used by the protocol, so we can use known lower bounds for AC^0 to prove deterministic lower bounds in the beeping model. We then use Razborov's Lemma, which shows that an AC^0 circuit can be approximated by a low-degree polynomial [35].

The beeping channel can be replaced by other types of channels: the degree bound of $O(r^3)$ in Theorem 1.1, and consequently the exponent in Corollary 1.2, change depending on the channel. For example, if we use a channel with collision detection, the degree bound changes to r^5 and the corresponding lower bound on PARITY_n and MAJORITY_n becomes $\Omega(n^{1/10})$.

1.1.2 Lower Bounds via Random Restrictions

While the technique of approximation by polynomials yields fairly general lower bounds that can be applied to a wide range of functions and models, it falls short of proving tight lower bounds, at least in the case of MAJORITY_n (and also for PARITY_n with randomized protocols); this is inherent, because these functions can be computed by polynomials of degree $\sqrt{n}$. We were especially interested in the MAJORITY function, because a lower bound on MAJORITY implies a lower bound on computing a sum; moreover, a lower bound on $\text{APPROX-MAJORITY}_{n,\epsilon}$, the promise problem of distinguishing whether $\sum_i x_i \geq (1/2 + \epsilon)n$ or whether $\sum_i x_i < (1/2 - \epsilon)n$, implies a lower bound on computing a $(1 \pm \epsilon)$-approximate sum. We were therefore especially motivated to prove tight bounds for MAJORITY_n and $\text{APPROX-MAJORITY}_{n,\epsilon}$.

To do so, we use another classical technique from circuit complexity, called *random restrictions* (defined in Subsection 4.2). Essentially, we show that for a certain class of functions, there is no "clever" way to use the beeping channel, and we cannot do much better than simply having the nodes speak one after the other. For this simpler setting we can then use known results.

Using this technique, we prove a nearly-tight lower bound on $\text{APPROX-MAJORITY}_{n,\epsilon}$, and consequently we obtain the same bound on $(1 \pm \epsilon)$-approximate sum. We mention that our proof also yields a similar lower bound for a "weaker" problem, called $\text{COIN}_{n,\epsilon}$, see Corollary 4.11.

▶ **Theorem 1.3.** *Every randomized beeping protocol that solves* Approx-Majority$_{n,\epsilon}$ *with error* $1/10$ *and* $\epsilon > c/\sqrt{n}$ *(for some sufficiently large constant* c*), must use* $\Omega\left(\frac{1}{\epsilon^2}\right)$ *rounds.*

The same theorem applies to other "simple" channels, such as the channel with collision detection.

▶ **Corollary 1.4.** *Any randomized beeping protocol for* Majority$_n$ *with error* $1/10$ *requires* $\Omega(n)$ *rounds.*

Our lower bound applies even if nodes have a shared source of randomness. It is easy to see that in this setting, our lower bound is *tight*: sampling $1/\epsilon^2$ random nodes, and having these nodes announce their inputs one after the other, solves Approx-Majority$_{n,\epsilon}$ with constant error. If nodes only have *private* randomness, the problem becomes more challenging, but [29] shows it can still be solved in $O(\log^2 /\epsilon^2)$ rounds. Therefore, our result is tight up to the polylogarithmic dependence on n even with private randomness.

We are also able to give a lower bound of $\Omega(1/\epsilon^2)$ on computing a $(1 \pm \epsilon)$-approximation to the size of the network in the beeping model and related models. This problem was studied in [10], where the authors give an algorithm that runs in $O(\log(1/\delta)/\epsilon^2 + \log\log n)$ rounds and succeeds with probability $1 - \delta$, and prove a lower bound of $\Omega(\log(1/\delta)/\epsilon + \log\log n)$. A lower bound of $\Omega(1/(\epsilon^2 \log(1/\epsilon)) + \log\log n)$ is shown in [15] for the related model of *RFID networks.* Computing an approximate sum reduces to the problem of approximating the size of the network, by simply having nodes with input 0 pretend that they are not present; thus, Theorem 1.3 also gives a lower bound of $\Omega(1/\epsilon^2)$ on this problem. The converse is not necessarily true; the lower bounds of [10, 15] rely on being able to choose which nodes participate in the computation, and they do not apply to computing an approximate sum in a fixed-size network. Our result recovers the dependence of $1/\epsilon^2$ in these bounds, and extends it to other models.

1.2 Related Work

The literature on wireless networks is vast; for lack of space, we survey only work that is directly related to out results. Many radio network algorithms use approximate counting as a subroutine, or solve it directly: [25, 13, 26, 27, 32] give constant-factor approximations in polylogarithmic time in the general radio model, under various assumptions, such as whether or not collision detection is present, and whether there is an adversary that can corrupt some messages.

In the beeping model [17], approximate counting was studied in [10], which gives an upper bound of $O(\log\log n + \log(1/\delta)/\epsilon^2)$, where ϵ is the approximation quality and δ is the error probability, and a lower bound of $\Omega(\log\log n + \log(1/\delta)/\epsilon)$. It is assumed in [17] that nodes do not have unique identifiers, and nothing is known a-priori about the size of the network. We show a lower bound of $\Omega(1/\epsilon^2)$ on the same problem, which holds even when nodes do have identifiers, and the size of the network is initially known up to a constant factor. In [29], the *approximate sums* problem is introduced: every node has some number in the range $\{0\} \cup [1, m]$, and each node must compute a $(1 \pm \epsilon)$-approximation to the sum of its neighbors' numbers. The authors give an algorithm for this problem that runs in $O((\log^2 + \log n \log m)/\epsilon^2)$ rounds, and use it to develop algorithms for PageRank and related problems. Our lower bound shows that the quadratic dependence on $1/\epsilon$ of the algorithm from [29] is inherent, even in single-hop networks when the inputs are Boolean, and even when the nodes have public randomness.

In [33], Newport gives a general technique for proving lower bounds in radio networks, and used it to prove and unify existing proofs for the wake-up and broadcast problems in a few different variants of the model (with or without collision detection and multiple channels). The focus in [33] is different than ours: we are mainly interested in the computational power inherent in the broadcast medium, while [33] gives a unified strategy for quantifying the cost of uncertainty and symmetry breaking.

The computational power of the beeping model is also studied in [21], but from a different perspective: [21] characterizes the number of states the nodes must have in order to solve randomized leader election, and show that with this number of states, it is also possible to simulate a logspace Turing machine with a constant number of unary input tapes. In a sense, [21] shows what the beeping model *can* do, while we focus on what it *cannot* do. Unlike [21], we do not restrict the computational power of the nodes themselves; they can have unbounded space or solve undecidable problems if they so wish.

A somewhat related model to radio networks is *RFID networks*, and many approximate counting protocols have been developed for that setting – we refer to [15] for a survey of this line of research. It is shown in [15] that RFID networks require $\Omega(\log\log n + 1/(\epsilon^2 \log(1/\epsilon)))$ to estimate their size to within $(1 \pm \epsilon)$, and this is almost tight.

2 Preliminaries

Notation. We use bold-face letters to denote random variables. All logarithms are base 2 (unless stated otherwise).

Given a string $s = s_1 \dots s_k \in \{0,1\}^*$ and an index $i \le k$, we let $s_{<i} = s_1 \dots s_{i-1}$ denote the length-$(i-1)$ prefix of s (with $s_{<1} = \varepsilon$, the empty string).

2.1 The Beeping Model

Model. In the single hop *beeping model*, a set of n nodes communicate over a shared channel. We assume that each node i has a single input bit, $x_i \in \{0,1\}$.

Communication occurs in synchronous rounds. In every such round, each node can choose to either beep or listen. If a node $i \in [n]$ listens in round m, it can only distinguish between silence (no other node beeps in round m) or the presence of one or more beeps (at least one other node beeps in round m). Thus, we can think of each node as broadcasting a bit in every round (where 0 corresponds to silence and 1 means beeping), and all nodes receive the OR of the n bits sent in this round.

Protocols. An r*-round deterministic protocol* for the beeping model specifies, for each node $i \in [n]$, two functions: a *broadcast function*, specifying whether node i should beep in each round $\ell = 1, \dots, r$, as a function of its input x_i and the transcript (i.e., the contents of the channel) in rounds $1, \dots, \ell - 1$; and an *output function*, which takes the complete r-round transcript and the input x_i of node i, and determines what value node i should output at the end of the protocol. Since we are concerned only with Boolean functions here, we assume for convenience that the protocol's output is the contents of the channel in the last round (i.e., if the transcript is $b_1, \dots, b_r$, then all nodes output b_r). We refer to r as the *length* or *running time* of the protocol.

A *randomized* beeping protocol $\mathcal{P}$ is a distribution over deterministic beeping protocols. Note that this definition corresponds to assuming that the nodes have *shared randomness*, an assumption that is usually avoided when designing beeping protocols, but this only strengthens our lower bounds.

The *length* of a randomized protocol $\mathcal{P}$ is the maximum length of a deterministic protocol in the support of $\mathcal{P}$.

3 Beeping Lower Bounds via Polynomial Approximations

We begin by showing that the beeping model is no more powerful than AC^0 *circuits* where deterministic protocols are concerned, and for randomized protocols, a beeping protocol can be *approximated by a low-degree polynomial.* For both computation models (AC^0 circuits and low-degree polynomials), powerful lower bounds are known, and we can then apply these lower bounds to the beeping model.

3.1 From Deterministic Beeping Protocols to Circuits

Fix an r-round deterministic protocol P for the beeping model. We would like to simulate P by a "simple" circuit C_P whose output agrees with P on all inputs $x \in \{0,1\}^n$.

The natural approach would be to try to simulate P round-by-round: in each round, every node decides whether or not to beep, as a function of the transcript (i.e., the contents of the channel) so far, and the contents of the channel is an OR over the nodes' decisions in the current round. Thus, an r-round deterministic protocol can be represented as a circuit of depth $O(r)$ and size $2^{O(r)}$, where the exponential size comes from the fact that each node's decision whether to beep in a given round can be an arbitrarily complex function of the transcript so far and its input. Unfortunately, no lower bounds are currently known against such circuits, except when r is very small (e.g., less than logarithmic in the input size).

Instead of constructing a circuit with high depth, given a deterministic beeping protocol, we transform it into a circuit of *constant depth*, composed of AND, OR and NOT gates with unbounded fan-in. We can then apply known AC^0 lower bounds against constant-depth circuits.

The circuit we construct is in negation-normal form, i.e., NOT gates are used only on input wires. To simplify the presentation, we assume that the circuit receives as input the n variables $x_1, \ldots, x_n$ on which it computes, as well as their negations, $\overline{x_1}, \ldots, \overline{x_n}$. The *depth* of the circuit is defined to be the maximum number of AND or OR gates on any path from an input (either x_i or $\overline{x_i}$) to the output of the circuit; the *size* of a circuit is the total number of wires.

▶ **Lemma 3.1.** *Let P be a deterministic beeping protocol between n nodes, each node $i \in [n]$ holding an input bit x_i. Assume that P has worst-case running time of r rounds. Then there exists a circuit C_P of depth 3 and size $2^{O(r+\log n)}$ such that $C_P(x) = P(x)$ for all $x \in \{0,1\}^n$.*

Proof. We construct C_P by "flattening" the protocol: instead of simulating it round-by-round, we *guess* an accepting transcript of the protocol (i.e., we guess the contents of the channel in every round, in an execution that leads to output 1), and verify that indeed the protocol would generate this transcript on the input at hand. Here, "guessing" corresponds to an OR over all accepting transcripts; verifying that a given transcript is consistent with the input can be done using a small depth-2 circuit.

Let us describe the construction more formally, from the bottom up.

Observe that for each player i, once we have fixed the transcript $t_{<\ell} \in \{0,1\}^{\ell-1}$ of the rounds preceding round ℓ, player i's decision whether to beep in round ℓ next round depends only on its input x_i. Thus, it is either constant or a literal, x_i or $\overline{x_i}$.

Let $D^i_{t,\ell}$ be the depth-0 circuit representing player i's decision whether to beep or not in round ℓ, when the transcript of the rounds up to ℓ is $t_{<\ell}$. (Again, $D_{t,\ell}$ is either constant or a literal). Our next step is to construct a circuit $R_{t,\ell}$ that "verifies" that round ℓ is consistent with the preceding rounds and the input: that is, $R_{t,\ell}$ outputs 1 on input x iff when P is

executed on input x, if the transcript of the first $\ell - 1$ rounds is $t_{<\ell}$, then that the contents of the channel in round ℓ is indeed t_ℓ. By definition, the contents of the beeping channel is a disjunction over the players' decisions, so we define:

$$R_{t,\ell} = \begin{cases} \bigvee_{i\in[n]} D^i_{t,\ell}, & \text{if } t_\ell = 1, \\ \bigwedge_{i\in[n]} \neg D^i_{t,\ell}, & \text{if } t_\ell = 0. \end{cases}$$

Note that in case $t_\ell = 0$, we want to verify that no player decided to beep, and since the inputs to the circuit are $x_1, \ldots, x_n, \overline{x_1}, \ldots, \overline{x_n}$, we can still verify this using a single AND gate. As a result, $R_{t,\ell}$ is always a single gate; it has depth 1 and size at most n.

Now, let

$$V_t = \bigwedge_{\ell=1}^{r} R_{t,\ell}$$

be a depth-2 circuit of size $O(n \cdot r)$ that checks whether the transcript t would indeed be generated on the current input and outputs 1 iff this is the case.

Finally, let $\mathcal{T} \subseteq \{0,1\}^r$ be the set of accepting transcripts of P (i.e., transcripts that cause the nodes to output 1). The circuit C_P is given by

$$C_P = \bigvee_{t\in\mathcal{T}} V_t,$$

a depth-3 circuit of size $O(n \cdot r \cdot 2^r) = 2^{O(r+\log n)}$, since $|\mathcal{T}| \leq 2^r$ and the size of each sub-circuit V_t is $O(n \cdot r)$. ◀

Using this transformation, we can immediately "import" known lower bounds for AC^0, and obtain, for example, lower bounds for the round complexity of beeping protocols for the Parity and Majority functions.

▶ **Corollary 3.2.** *Any deterministic beeping protocol that computes* $\textsc{Parity}_n$ *or* $\textsc{Majority}_n$ *requires* $\Omega(\sqrt{n})$ *rounds.*

Proof. It is known that any depth-3 circuit for $\textsc{Parity}_n$ or $\textsc{Majority}_n$ must have size $2^{\Omega(\sqrt{n})}$ [22]. Together with Lemma 3.1, we obtain the corresponding lower bounds for the beeping model. ◀

We remark that while every r-round beeping protocol can be simulated by an AC^0 circuit of depth 3 and size $2^{O(r+\log n)}$ (Lemma 3.1), the converse is not true: AC^0 circuits have inherent parallelism, which makes them more powerful than the beeping model. One example is the majority function, which can be computed by an AC^0 circuit of depth 3 and size $2^{O(\sqrt{n})}$, but requires $\Omega(n)$ rounds in the beeping model (as we show in the next section). An example that seems even worse is the function

$$\textsc{Tribes}_{k,\ell}(x_1, \ldots, x_{k\cdot\ell}) = \bigvee_{i=1}^{k} \left(\bigwedge_{j=1}^{\ell} x_{i,j} \right).$$

For any $k, \ell \in \mathbb{N}$, the function $\textsc{Tribes}_{k,\ell}$ is computed by an AC^0 circuit of depth 2 and size $k \cdot \ell$. However, it seems plausible that the beeping model requires $\Omega(k)$ rounds to solve $\textsc{Tribes}_{k,\ell}$: even though each inner conjunction can be computed in a single round, we still need to compute k such conjunctions, and it seems this should require $\Omega(k)$ rounds (but we have not proven this intuition).

3.2 Randomized Beeping Protocols

The lower bound technique above applies only to deterministic protocols, since it relies on deterministic lower bounds for AC^0. To extend this approach to randomized protocols, we take it one step further, and use the fact that a small low-depth circuit can be *approximated by a low-degree polynomial*, while "complex" functions like Parity and Majority cannot be approximated by a low-degree polynomial. This is one central approach used to prove AC^0 lower bounds, e.g., [35, 38], and we use the construction from [35]:

▶ **Lemma 3.3** (Razborov's Lemma, [35]). *Given a circuit C of size s and depth d, for any sufficiently small $\epsilon \in (0, 1/2)$, there exists a distribution $\mathcal{G}$ over multivariate polynomials $g(x_1, \dots, x_n) \in \mathbb{F}_2[X_1, \dots, X_n]$ of degree at most $(\log(s/\epsilon))^d$, such that for all $x \in \{0,1\}^n$,*

$$\Pr_{\boldsymbol{g} \sim \mathcal{G}} [\boldsymbol{g}(x) \neq C(x)] \leq \epsilon.$$

Here, $\mathbb{F}_2[X_1, \dots, X_n]$ denotes the polynomials over $x_1, \dots, x_n$ with coefficients in $\mathbb{F}_2$.

Combining Lemma 3.3 with Lemma 3.1 yields the following corollary:

▶ **Corollary 3.4** (Theorem 1.1 restated). *If there is a randomized r-round beeping protocol $\mathcal{P}$ that computes $f : \{0,1\}^n \to \{0,1\}$ with error ϵ, then there is a polynomial $g \in \mathbb{F}_2[X_1, \dots, X_n]$ of degree $O((r + \log n)^3)$ which agrees with f on all but a 2ϵ-fraction of the inputs in $\{0,1\}^n$.*

Now we can rely on the fact that Majority and Parity do not have low-degree approximating polynomials:

▶ **Theorem 3.5** ([35, 38]). *For any polynomial $p \in \mathbb{F}_2[X_1, \dots, X_n]$ of degree t,*

$$\Pr_{\boldsymbol{x} \sim \mathsf{U}(\{0,1\}^n)} [p(\boldsymbol{x}) = \textsc{Majority}_n(\boldsymbol{x})] \leq \frac{1}{2} + O\left(t/\sqrt{n}\right),$$

and similarly for $\textsc{Parity}_n$.

In other words, to obtain *constant* error on the Majority or Parity function, our polynomial must have degree $t = \Omega(\sqrt{n})$. Therefore:

▶ **Corollary 3.6** (Corollary 1.2 restated). *Any randomized beeping protocol for $\textsc{Parity}_n$ or $\textsc{Majority}_n$ with error $1/10$ requires $\Omega(n^{1/6})$ rounds.*

Proof of Corollary 1.2. Given $\mathcal{P}$, we construct for each deterministic protocol P in the support of $\mathcal{P}$ the corresponding circuit C_P from Lemma 3.1, which agrees with P on all inputs. Each C_P has depth 3 and size at most $2^{O(r+\log n)}$, so using Razborov's Lemma (Lemma 3.3), there is a distribution $\mathcal{G}_P$ on polynomials of degree at most

$$\left(\log \frac{2^{O(r+\log n)}}{\epsilon}\right)^3 = O((r + \log n)^3),$$

such that for all $x \in \{0,1\}^n$,

$$\Pr_{\boldsymbol{g} \sim \mathcal{G}_P} [\boldsymbol{g}(x) \neq P(x)] \leq \epsilon.$$

Now let $\mathcal{G}$ be the distribution over $\mathbb{F}_2[X_1, \dots, X_n]$ where we first pick $\boldsymbol{P} \sim \mathcal{P}$, and then sample $\boldsymbol{g} \sim \mathcal{G}_{\boldsymbol{P}}$. Then, for any $x \in \{0,1\}^n$,

$$\begin{aligned}\Pr_{\boldsymbol{g} \sim \mathcal{G}} [\boldsymbol{g}(x) \neq f(x)] &\leq \Pr_{\boldsymbol{P} \sim \mathcal{P}, \boldsymbol{g} \sim \mathcal{G}_{\boldsymbol{P}}} [\boldsymbol{P}(x) \neq \boldsymbol{g}(x)] + \Pr_{\boldsymbol{P} \sim \mathcal{P}} [\boldsymbol{P}(x) \neq f(x)] \\ &\leq \epsilon + \epsilon = 2\epsilon.\end{aligned}$$

Viewed another way: when we sample $\boldsymbol{x}$ uniformly at random from $\{0,1\}^n$,

$$\mathop{\mathrm{E}}_{\boldsymbol{g}\sim\mathcal{G}}\left[\Pr_{\boldsymbol{x}\sim\mathsf{U}(\{0,1\}^n)}[\boldsymbol{g}(\boldsymbol{x})\neq f(\boldsymbol{x})]\right]\leq 2\epsilon.$$

Therefore, there exists at least one polynomial g in the support of $\mathcal{G}$ such that

$$\Pr_{\boldsymbol{x}\sim\mathsf{U}(\{0,1\}^n)}[g(\boldsymbol{x})\neq f(\boldsymbol{x})]\leq 2\epsilon,$$

and this polynomial, like all polynomials in the support of $\mathcal{G}$, has degree $O((r+\log n)^3)$. ◀

Unfortunately, this is more or less as far as we can go using this approach: there exists a circuit of depth 3 size $2^{O(\sqrt{n})}$ for Majority, and if we relax the requirement to computing Majority correctly on a $(1-\epsilon)$-fraction of inputs (instead of all inputs), the circuit size reduces to $2^{O(n^{1/4})}$ [6]. Therefore, our approach cannot yield a lower bound better than $\Omega(\sqrt{n})$ for deterministic beeping protocols, or better than $\Omega(n^{1/4})$ for randomized protocols, even if we somehow improved the degree of the approximating polynomial in our construction. In the next section, we show that by applying the technique of *random restrictions* directly to a beeping protocol, we can obtain a lower bound of $\tilde{\Omega}(n)$ for Majority_n (and for Parity_n), and we can also handle $\text{Approx-Majority}_{n,\epsilon}$.

4 Beeping Lower Bounds via Random Restrictions

In this section we use another central technique from circuit complexity, called *random restrictions*, to show that the beeping model cannot compute certain functions efficiently. For lack of space, some proofs are omitted.

Random restriction were used in the seminal proof that AC^0 circuits cannot compute the parity function [5, 20, 40, 22], and since then have found many applications. The basic idea is the following: we are given a Boolean function $f:\{0,1\}^n\to\{0,1\}$, and a circuit (or decision tree) C that computes f. We would like to show that C must have *high depth.* To do this, we randomly choose a subset of the input variables, and fix their values to 0 or 1 at random; this is referred to as "hitting the circuit with a random restriction". We say that an input variable "survived" the restriction if it was not fixed. Then we show:

(a) The complexity of the circuit C is significantly reduced. For example, after hitting an AC^0 circuit with a random restriction where every variable survives with some inverse polynomial probability, the circuit becomes *constant* with high probability.

(b) The complexity of the function f is *not* significantly reduced. For example, hitting the parity function with a random restriction yields a parity over the set of inputs we did not fix.

We begin by describing OR decision trees, a convenient way to represent beeping protocols. Then we formally define random restrictions and analyze what happens to an OR decision tree when hit with a random restriction; finally, we use this machinery to prove a lower bound for approximate majority.

4.1 OR Decision Trees

When each node has a single-bit input $x_i\in\{0,1\}$, a deterministic beeping protocol can be modeled as an *OR decision tree*:

▶ **Definition 4.1** (OR decision tree). *An* OR decision tree of depth d *on variables* $x_1, \ldots, x_n$ *is a binary tree* T *of depth* d*, where each inner node* $v \in \{0,1\}^*$ *is labeled with a disjunction* $D_v = \bigvee_{i=1}^{w_i} \ell_v^i$*. Here, each* ℓ_v^i *is a literal,* $\ell_v^i \in \{x_i, \overline{x_i} : i \in [n]\} \cup \{0,1\}$*. The leafs of the tree are labeled with Boolean values.*

The value *of* T *on an input* $x = (x_1, \ldots, x_n) \in \{0,1\}^n$ *is defined by induction on the height of the tree: the value of a leaf is its label; the value of an inner node* v *is the value of its left subtree if* $D_v(x) = 0$*, or the value of its right subtree if* $D_v(x) = 1$*.*

OR decision trees are a generalization of *decision trees*, where every node queries a single variable (see [12] for a survey). Other generalizations have been considered, *e.g., Parity Decision Trees* (see [41] for a survey). To our knowledge, however, OR decision trees have not been studied before in related contexts.

Beeping protocols as OR trees. To represent a deterministic beeping protocol P as an OR decision tree, we construct the following tree: at each node $v \in \{0,1\}^*$, we place the disjunction $D_v = \bigvee_{i=1}^{n} \ell_v^i$, where $\ell_v^i = P_i(v)$ is 1 if node i beeps after witnessing the transcript v, and 0 otherwise. At each leaf we write the output of P on the corresponding transcript.

A *randomized* beeping protocol is simply a distribution over OR decision trees.

Observe that the depth of the OR decision tree representing a beeping protocol is the number of rounds used by the protocol. Therefore, to prove that a function f does not have a deterministic beeping protocol using r rounds, it suffices to show that every OR decision tree that computes f has depth greater than r, and similarly, for randomized protocols, we must show that every distribution over OR decision trees that computes f with low error has some tree in its support with depth greater than r.

In the sequel, we restrict attention to OR decision trees where at each node of the tree, no variable is queried more than once; that is, the tree does not contain a disjunction of the form $x_i \vee x_i \vee \ldots$ or of the form $x_i \vee \overline{x_i} \vee \ldots$. This is true of trees generated from beeping protocols, but we can also assume it, without loss of generality, about general trees – in the first case we can simply get rid of duplicates, and in the second case the value of the node is always 1, and we can remove it from the tree and replace it with its right subtree.

4.2 Random Restrictions

Let us review the formal definition of a random restriction, and study what happens to an OR decision tree when we hit it with a random restriction.

▶ **Definition 4.2** (Restriction). *An* n*-bit* restriction *is a mapping* $r : \{x_1, \ldots, x_n\} \to \{0, 1, *\}$*. Given an input* $x \in \{0,1\}^n$*, the* restriction *of* x *to* r*, denoted* $x|_r \in \{0,1\}^n$*, is defined by*

$$(x|_r)_i = \begin{cases} x_i & \textit{if } r(x_i) = *, \\ r(x_i) & \textit{otherwise.} \end{cases}$$

Given a function $f : \{0,1\}^n \to \{0,1\}$*, the* restriction of f to r*, denoted* $f|_r : \{0,1\}^n \to \{0,1\}$*, is defined by* $f|_r(x) = f(x|_r)$*.*

For a parameter $\alpha \in [0,1]$, let $\mathcal{D}_{n,\alpha}$ be the distribution over n-bit restrictions obtained by setting, independently for each $i \in [n]$,

$$r(x_i) = \begin{cases} * & \text{w.p. } 1-\alpha, \\ 0 & \text{w.p. } \alpha/2, \\ 1 & \text{w.p. } \alpha/2. \end{cases}$$

In other words, each input variable survives with probability $1-\alpha$, and otherwise it is fixed to 0 or 1 with equal probability.

What happens to an OR decision tree when we hit it with a random restriction? We show that for any path of length d, with high probability, the *total* number of variables queried along the path drops to $\tilde{O}(d)$. (Recall that even a single node of an OR tree may be labeled by a disjunction of all the variables, so this is a significant.) To show this, let us first formally define the "total cost" of a path, and then study in turn what happens to a disjunction, to a path, and to the entire tree when we hit them with a random restriction.

The total cost of a path. Let π be a path – a sequence of disjunctions, $\pi = D_1, \dots, D_d$, where $D_i = \bigvee_{j=1}^{w_i} \ell_i^j$ for each i, and each ℓ_i^j is a literal. The *cost* of each disjunction D_i is its width, $\text{cost}(D_i) = w_i$. Define the *total cost* of π as

$$\text{cost}(\pi) = \sum_{i=1}^{d} \text{cost}(D_i).$$

The number of variables queried along π is *at most* $\text{cost}(\pi)$, although it could be smaller, if the same variable is queried by more than one disjunction along π.

Hitting a disjunction with a random restriction. When we hit a disjunction $D = \bigvee_{i=1}^{w} \ell_i$ of width w with a restriction r, the resulting function is also a disjunction: if there is some literal ℓ_i in D that is fixed to 1 (i.e., $\ell_i|_r = 1$), then the value of D becomes fixed, $D|_r = 1$. Otherwise, $D|_r = \bigvee_{i \in S_r} \ell_i$, where $S_r = \{j \in [w] : \ell_j|_r = \ell_j\}$. In this case we say that D *survives* r.

We have

$$\Pr_{\boldsymbol{r} \sim \mathcal{D}_{n,\alpha}} [D \text{ survives } \boldsymbol{r}] = \left(1 - \frac{\alpha}{2}\right)^w,$$

so wide disjunctions have low survival probability. (Recall that we assumed D does not query the same variable more than once, and therefore, the value of each literal after hitting it with r is independent of the other literals.)

Hitting a path with a random restriction. Consider a path $\pi = D_1, \dots, D_d$. Given a restriction r, let $\pi|_r = D_1|_r, \dots, D_d|_r$ be the path obtained by hitting each node (disjunction) of π with r.

When we apply a random restriction to π, it has the effect of "killing off" wide disjunctions, and therefore, we expect the total cost of the path to be fairly small:

▶ **Lemma 4.3.** *For any path $\pi = D_1, \dots, D_d$,*

$$\mathop{\mathrm{E}}_{\boldsymbol{r} \sim \mathcal{D}_{n,1/2}} [\text{cost}(\pi|_{\boldsymbol{r}})] \le 2|\pi|.$$

Proof. Let w_i be the width of D_i for $i \in [d]$. Since a disjunction of width w survives with probability $(3/4)^w$, the expected cost of $\pi|_{\boldsymbol{r}}$ is

$$\begin{aligned}\mathop{\mathrm{E}}_{\boldsymbol{r}\sim\mathcal{D}_{n,1/2}}[\mathrm{cost}(\pi|_{\boldsymbol{r}})] &= \sum_{i=1}^{|\pi|} \mathop{\mathrm{E}}_{\boldsymbol{r}\sim\mathcal{D}_{n,1/2}}[\mathrm{cost}(D_i|_{\boldsymbol{r}})] \\ &< \sum_{i=1}^{|\pi|}\left(w_i \cdot \mathop{\Pr}_{\boldsymbol{r}\sim\mathcal{D}_{n,1/2}}[D_i \text{ survives } \boldsymbol{r}]\right) \\ &= \sum_{i=1}^{|\pi|}\left(w_i\left(\frac{3}{4}\right)^{w_i}\right) < 2|\pi|.\end{aligned}$$

In the last step we used the fact that $(3/4)^x \cdot x < 2$ for all $x \geq 1$. ◀

Hitting the entire tree with a random restriction. For an OR decision tree T and an input $x \in \{0,1\}^n$, let $\pi_T(x)$ be the computation path of T on x. We let $T|_r$ denote the tree obtained from T by replacing each disjunction D_v with the disjunction $D_v|_r$. Observe that if $f : \{0,1\}^n \to \{0,1\}$ is the function computed by T and r is a restriction, then $f|_r$ is the function computed by $T|_r$.

As we saw above, hitting T with a random restriction has the effect of reducing the cost of each path, with high probability. However, some small fraction of paths may retain high total cost (in fact, this is likely). We deal with these paths by *truncating* them.

► **Definition 4.4.** *Given a restriction r, an OR decision tree T and a cost bound $c \in \mathbb{N}$, we define a* truncated *OR decision tree $T|_{r,c}$ as follows: $T|_{r,c}$ is the same as $T|_r$, except that for any node v at depth c, if v is not a leaf, we replace v with a leaf labeled with 0.*

Truncating an OR decision tree can increase its error, but if we choose the depth bound appropriately, the error does not increase by much:

▷ Claim 4.5. Let T be an OR decision tree of depth d on n inputs. Let $\eta \in (0, 1/2)$, and let $\gamma = 2/\eta$. Then for any input $x \in \{0,1\}^n$, we have

$$\mathop{\Pr}_{\boldsymbol{r}\sim\mathcal{D}_{n,1/2}}[T|_{\boldsymbol{r},\gamma d}(x|_{\boldsymbol{r}}) \neq T|_{\boldsymbol{r}}(x|_{\boldsymbol{r}})] \leq \eta.$$

Proof. Let π be the computation path of T on x. By Lemma 4.3 and Markov,

$$\mathop{\Pr}_{\boldsymbol{r}\sim\mathcal{D}_{n,1/2}}[\mathrm{cost}(\pi|_{\boldsymbol{r}}) > (2/\eta)|\pi|] < \eta.$$

Thus, the probability that π needs to be truncated in $T|_{\boldsymbol{r},\gamma d}$ is at most η. If π is not truncated, then it is the same as in $T|_{\boldsymbol{r}}$, and in particular it returns the same answer as $T|_{\boldsymbol{r}}$. Otherwise, we may err, but this happens with probability at most η. ◁

Finally, we observe that an OR decision tree with *low total cost* can be "unrolled" into a *plain* decision tree of *low depth*: we can replace every disjunction of width w by a plain decision tree of depth w that queries the same variables and computes their OR. (In general, any function on w variables can be computed by a decision tree of depth w.)

▷ Claim 4.6. Let T be an OR decision tree where all paths have total cost at most d. Then there exists a (plain) decision tree of depth d that computes the same function as T.

It is known that plain decision trees require high depth to compute an approximate majority; next, we show how we can apply this result to obtain a lower bound for the beeping model.

4.3 The Coin Problem

The *coin problem* [37, 1, 11, 39, 16] is essentially a randomized version of *approximate majority*, and it is often used to prove lower bounds on approximate majority.

In the coin problem, $\textsc{Coin}_{n,\epsilon}$, we have an ϵ-biased coin, where either "heads" or "tails" has probability $1/2+\epsilon$, but we do not know which. The coin is flipped n independent times, and given the n outcomes we need to decide if the coin is biased towards heads or tails. More formally:

▶ **Definition 4.7** (The coin problem). *We say that a function $f : \{0,1\}^n \to \{0,1\}$* solves the coin problem $\textsc{Coin}_{n,\epsilon}$ *with error δ if, for any $a \in \{0,1\}$,*

$$\Pr_{\boldsymbol{x}\sim \mathsf{B}(1/2+(-1)^a\cdot\epsilon)^n}[f(\boldsymbol{x}) = a] > 1-\delta.$$

We say that a *distribution* $\mathcal{F}$ over functions $f : \{0,1\}^n \to \{0,1\}$ solves $\textsc{Coin}_{n,\epsilon}$ with error δ if

$$\Pr_{\boldsymbol{x}\sim \mathsf{B}(1/2+(-1)^a\cdot\epsilon)^n, \boldsymbol{f}\sim\mathcal{F}}[\boldsymbol{f}(\boldsymbol{x}) = a] > 1-\delta.$$

Abusing the terminology slightly, we will also say that a distribution $\mathcal{T}$ over OR decision trees solves $\textsc{Coin}_{n,\epsilon}$ if the distribution of *functions computed by trees sampled from* $\mathcal{T}$ solves $\textsc{Coin}_{n,\epsilon}$.

It is not hard to see that the best strategy for solving the coin problem is to output the *majority* of the n inputs. By Chernoff, this strategy works as long as $\epsilon > c/\sqrt{n}$, where $c = c(\delta)$ depends on the desired error probability δ. (If $\epsilon \ll c/\sqrt{n}$, it is impossible to solve $\textsc{Coin}_{n,\epsilon}$ with error δ.) Moreover, when $\epsilon > c/\sqrt{n}$ for some sufficiently large $c = c(\delta)$, it suffices to compute an $\epsilon/2$-*approximate majority*: the probability that $|\sum_i \boldsymbol{x}_i - n/2| \leq \epsilon/2$ is small, so even an $\epsilon/2$-approximate majority will yield the right answer w.h.p. Thus, a lower bound on solving $\textsc{Coin}_{n,\epsilon}$ immediately yields a lower bound on computing an $\epsilon/2$-approximate majority on n bits (with slightly higher error).

It is known that the coin problem is very difficult for regular decision trees:

▶ **Theorem 4.8** ([16]). *There is a constant $c > 0$ such that for any $\epsilon > c/\sqrt{n}$, the depth of any decision tree that solves $\textsc{Coin}_{n,\epsilon}$ with error $1/10$ is $\Omega(1/\epsilon^2)$.*

Next, we will show that this is also true for OR decision trees (up to constants).

A key property of the coin problem is that when we hit it with a random restriction, it does not become much easier: essentially, fixing a random subset of the outcomes of the coin to 0 or 1 corresponds to *increasing the bias of the coin*, but not by much. This property is used to prove lower bounds for AC^0 using the coin problem.

▶ **Lemma 4.9** ([11, 16]). *Let $\alpha \in [0,1)$. Suppose that $f : \{0,1\}^n \to \{0,1\}$ solves the coin problem $\textsc{Coin}_{n,\epsilon}$ with error δ. The distribution $f|_{\boldsymbol{r}}$ over functions, where $\boldsymbol{r} \sim \mathcal{D}_{n,\alpha}$, solves $\textsc{Coin}_{n,\frac{\epsilon}{1-\alpha}}$ with error δ.*

We showed that when hit with a random restriction, an OR decision tree of depth d "collapses" to a plain decision tree of roughly the same depth. The coin problem, on the other hand, does not become significantly easier when hit with a random restriction (Lemma 4.9). Combining everything, we have:

▶ **Theorem 4.10.** *Let $\mathcal{T}$ be a distribution over OR decision trees that solves $\textsc{Coin}_{n,\epsilon}$ with error at most $1/10$ and $\epsilon > c/\sqrt{n}$ (for some large enough constant c). Then, there is a tree T in the support of $\mathcal{T}$ that has depth $\Omega(1/\epsilon^2)$.*

As we showed in Section 4.1, a beeping protocol can be modeled as an OR decision tree, whose depth corresponds to the number of rounds of the protocol. We therefore obtain the following corollary, which implies Theorem 1.3:

▶ **Corollary 4.11.** *Any randomized beeping protocol that solves* $\text{Coin}_{n,\epsilon}$ *with error* $1/10$ *and* $\epsilon > c/\sqrt{n}$ *(for some large enough constant* c*), has* $\Omega(1/\epsilon^2)$ *rounds.*

5 Extension to Other Channel Types

In addition to the beeping model, our techniques can be used to prove lower bounds for other types of wireless channels.

Throughout the paper we have simulated a round of a beeping protocol by a circuit of depth one consisting of a single OR gate. When considering a different wireless channel $\mathcal{C}$, as long as this channel is not too complex, we can also model it as a "simple" circuit C. Then,

- If C is a relatively shallow circuit, then the techniques of Section 3 can be generalized to $\mathcal{C}$, by replacing the OR gates used to simulate one round of the beeping protocol by C-type circuits (see, e.g., Lemma 3.1).
- If hitting C with a random restriction (say, with constant survival probability) yields a circuit that depends on only a small number of inputs w.h.p. then the techniques of Section 4 can also be generalized to $\mathcal{C}$, by replacing the OR trees used to model a beeping protocol by C-trees (trees where each node is labeled by a C-type circuit).

The collision detection channel. Consider, for example, a wireless channel $\mathcal{C}$ with *collision detection*. The output of the channel is one of four symbols, $\bot, \top, 0$ and 1: if no node decides to broadcast, all nodes receive $\bot$; if more than one node decides to broadcast, all nodes receive $\top$; and if exactly one node decides to broadcast, all nodes receive the message it sent.

Next, we implement the operation of $\mathcal{C}$ as a simple circuit C. Note that unlike the beeping channel, in the collision detection channel, each node has two decisions to make: first, whether to broadcast; and second, what value to broadcast (if broadcasting). Accordingly, our circuit C will have inputs $x_1, \ldots, x_n, b_1, \ldots, b_n$, where x_i indicates whether node i broadcast ($x_i = 1$ means that node i did broadcast), and b_i gives the bit broadcast by node i in the case that it did broadcast. The circuit C outputs three bits s, c and b. If $s = 1$, then this is a silent round, i.e., $\mathcal{C}$ outputs $\bot$. If $c = 1$, then a collision has occurred, i.e., $\mathcal{C}$ outputs $\top$. Otherwise, if $s = c = 0$, then the output of $\mathcal{C}$ is the bit b. (We make sure that only one of these three cases holds.)

The circuit C consists of three parts, where each part computes one of c, s and b:

Computing s: the circuit C computes s by taking $s = \bigwedge_{i=1}^{n} \neg x_i$.
Computing c: the circuit C computes c by taking $c = \bigvee_{i \neq j} (x_i \wedge x_j)$.
Computing b: the circuit C computes b by taking $b = \bigvee_{i=1}^{n} (b_i \wedge x_i)$. Note that when $s = c = 0$, there is exactly one node i with $x_i = 1$, and in this case $b_i = \bigvee_{i=1}^{n} (b_i \wedge x_i)$.

The circuit C we constructed is of depth 2 and size $O(n^2)$. We could use it to construct an AC^0 circuit as we did in Lemma 3.1, and the resulting circuit would have depth 4 and size $2^{O(r+\log n)}$. However, we can do better using random restrictions. Instead of explicitly computing the survival probability of each node in the tree, we can appeal to Håstad's Switching Lemma, which analyzes the behavior of DNFs under random restrictions.

A *DNF of width* w is a formula of the form $\bigvee_{i=1}^{m} \left(\bigwedge_{j=1}^{k_i} \ell_{i,j} \right)$, where each $\ell_{i,j}$ is a literal, and $k_i \leq w$ for each $1 \leq i \leq m$. Let $\mathrm{DT}(f)$ denote the minimum depth of a (plain) decision tree that computes f.

▶ **Lemma 5.1** (Håstad's Switching Lemma [22]). *Let f be a DNF of width w over n variables, and let $\alpha \le 1/5$. Then for any $d \ge 0$,*

$$\Pr_{\boldsymbol{r}\sim\mathcal{D}_\alpha}[\mathrm{DT}(f|_{\boldsymbol{r}}) \ge d] \le (5\alpha w)^d.$$

In particular, whenever $5\alpha w \le 1/2$, we have $\mathrm{E}_{\boldsymbol{r}\sim\mathcal{D}_\alpha}[\mathrm{DT}(f|_{\boldsymbol{r}})] \le \sum_{d=0}^{\infty}(5\alpha w)^d \le 2$. The same holds for CNFs (circuits of the form $\bigwedge_{i=1}^{m}\bigvee_{j=1}^{k_i}\ell_{i,j}$).

In the circuit C that describes the behavior of the collision detection channel, each of the three parts is a DNF or a CNF of width ≤ 2. The Switching Lemma shows that if we hit a C-decision tree with a random restriction $\boldsymbol{r}$ where every variable survives with sufficiently small constant probability (e.g., $1/100$), and then "unroll" each C-type node into a decision tree with the smallest depth possible, then for each path π in the original C-decision tree, the expected length of the "unrolled" $\pi|_{\boldsymbol{r}}$ is $O(|\pi|)$. From here, we can proceed exactly as in Section 4, and obtain that the channel with collision detection also requires $\Omega(1/\epsilon^2)$ rounds to solve ϵ-approximate majority.

The additive channel. Another interesting example is the additive channel [14], where in every round, each of the n nodes broadcasts a bit, and each node hears the XOR (or, PARITY) of the broadcast bits.

It is known that the PARITY function is not very useful when it comes to computing MAJORITY: for example, any depth-3 AC^0 circuit that is additionally equipped with unbounded fan-in PARITY gates must still have size at least $2^{\Omega(n^{1/4})}$ to compute MAJORITY_n [38]. Thus, following the outline from Section 3, we see that a deterministic protocol for the additive single-hop network requires $\Omega(n^{1/4})$ rounds to compute MAJORITY_n.

Similarly to our approach in Section 4, one can model protocols over the additive channel as *parity decision trees*, objects that have been extensively studied (see [41] for a survey). Known lower bounds on the depth of a parity decision tree that computes a specific function, imply the same lower bound in the additive channel model (e.g., the lower bound on the parity decision tree complexity of the recursive majority function in [9]).

References

1. Scott Aaronson. BQP and the polynomial hierarchy. In *Symposium on Theory of Computing (STOC)*, pages 141–150, 2010.
2. Yehuda Afek, Noga Alon, and Ziv Bar-Joseph. Beeping an MIS. *Manuscript*, 2011.
3. Yehuda Afek, Noga Alon, Ziv Bar-Joseph, Alejandro Cornejo, Bernhard Haeupler, and Fabian Kuhn. Beeping a maximal independent set. *Distributed Computing*, 26(4):195–208, 2013.
4. Yehuda Afek, Noga Alon, Omer Barad, Eran Hornstein, Naama Barkai, and Ziv Bar-Joseph. A biological solution to a fundamental distributed computing problem. *Science*, 331(6014):183–185, 2011.
5. Miklós Ajtai. Σ_1^1-formulae on finite structures. *Annals of pure and applied logic*, 24(1):1–48, 1983.
6. Kazuyuki Amano. Bounds on the Size of Small Depth Circuits for Approximating Majority. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 59–70, 2009.
7. Reuven Bar-Yehuda, Oded Goldreich, and Alon Itai. On the Time-Complexity of Broadcast in Multi-hop Radio Networks: An Exponential Gap Between Determinism and Randomization. *J. Comput. Syst. Sci.*, 45(1):104–126, 1992.
8. Boaz Barak, Mark Braverman, Xi Chen, and Anup Rao. How to Compress Interactive Communication. *SIAM J. Comput.*, 42(3):1327–1363, 2013.

9 Eric Blais, Li-Yang Tan, and Andrew Wan. An inequality for the Fourier spectrum of parity decision trees. *CoRR*, abs/1506.01055, 2015. arXiv:1506.01055.

10 Philipp Brandes, Marcin Kardas, Marek Klonowski, Dominik Pajkak, and Roger Wattenhofer. Approximating the Size of a Radio Network in Beeping Model. In Jukka Suomela, editor, *Structural Information and Communication Complexity*, pages 358–373, 2016.

11 Joshua Brody and Elad Verbin. The Coin Problem and Pseudorandomness for Branching Programs. In *Symposium on Foundations of Computer Science (FOCS)*, pages 30–39, 2010.

12 Harry Buhrman and Ronald de Wolf. Complexity measures and decision tree complexity: a survey. *Theor. Comput. Sci.*, 288(1):21–43, 2002.

13 Ioannis Caragiannis, Clemente Galdi, and Christos Kaklamanis. Basic Computations in Wireless Networks. In *Algorithms and Computation*, 2005.

14 Keren Censor-Hillel, Bernhard Haeupler, Nancy A. Lynch, and Muriel Médard. Bounded-Contention Coding for the additive network model. *Distributed Computing*, 28(5):297–308, 2015.

15 Binbin Chen, Ziling Zhou, and Haifeng Yu. Understanding RFID Counting Protocols. In *Annual International Conference on Mobile Computing and Networking (MobiCom)*, pages 291–302, 2013.

16 Gil Cohen, Anat Ganor, and Ran Raz. Two Sides of the Coin Problem. In *Approximation, Randomization, and Combinatorial Optimization (APPROX/RANDOM)*, pages 618–629, 2014.

17 Alejandro Cornejo and Fabian Kuhn. Deploying Wireless Networks with Beeps. In *International Conference on Distributed Computing (DISC)*, pages 148–162, 2010.

18 Fabien Dufoulon, Janna Burman, and Joffroy Beauquier. Beeping a Deterministic Time-Optimal Leader Election. In *International Symposium on Distributed Computing (DISC)*, pages 20:1–20:17, 2018.

19 Klaus-Tycho Förster, Jochen Seidel, and Roger Wattenhofer. Deterministic Leader Election in Multi-hop Beeping Networks. In *International Symposium on Distributed Computing (DISC)*, pages 212–226, 2014.

20 Merrick L. Furst, James B. Saxe, and Michael Sipser. Parity, Circuits, and the Polynomial-Time Hierarchy. *Mathematical Systems Theory*, 17(1):13–27, 1984.

21 Seth Gilbert and Calvin Newport. The Computational Power of Beeps. In *Distributed Computing*, pages 31–46, 2015.

22 John Hastad. Almost Optimal Lower Bounds for Small Depth Circuits. *Advances in Computing Research*, 5:143–170, 1989.

23 Stephan Holzer and Nancy A. Lynch. Brief announcement: Beeping a maximal independent set fast, 2017.

24 Bojun Huang and Thomas Moscibroda. Conflict Resolution and Membership Problem in Beeping Channels. In *International Symposium on Distributed Computing (DISC)*, pages 314–328, 2013.

25 Tomasz Jurdziński, Mirosław Kutyłowski, and Jan Zatopiański. Energy-Efficient Size Approximation of Radio Networks with No Collision Detection. In *Computing and Combinatorics*, pages 279–289, 2002.

26 Jedrzej Kabarowski, Mirosław Kutyłowski, and Wojciech Rutkowski. Adversary Immune Size Approximation of Single-Hop Radio Networks. In *Theory and Applications of Models of Computation*, 2006.

27 Marek Klonowski and Kamil Wolny. Immune Size Approximation Algorithms in Ad Hoc Radio Network. In *Wireless Sensor Networks*, 2012.

28 Gillat Kol, Rotem Oshman, and Dafna Sadeh. Interactive Compression for Multi-Party Protocol. In *International Symposium on Distributed Computing (DISC)*, pages 31:1–31:15, 2017.

29 Zhiyu Liu and Maurice Herlihy. Approximate Local Sums and Their Applications in Radio Networks. In *Distributed Computing*, pages 243–257, 2014.

30 Yves Métivier, John Michael Robson, and Akka Zemmari. On Distributed Computing with Beeps. *CoRR*, abs/1507.02721, 2015. `arXiv:1507.02721`.

31 Saket Navlakha and Ziv Bar-Joseph. Distributed information processing in biological and computational systems. *Communications of the ACM*, 58(1):94–102, 2015.

32 Calvin Newport and Chaodong Zheng. Approximate Neighbor Counting in Radio Networks. In *International Conference on Principles of Distributed Systems (OPODIS)*, pages 26:1–26:16, 2018.

33 Calvin C. Newport. Radio Network Lower Bounds Made Easy. In *International Conference on Distributed Computing (DISC)*, pages 258–272, 2014.

34 David Peleg. Time-Efficient Broadcasting in Radio Networks: A Review. In *Distributed Computing and Internet Technology (ICDCIT)*, pages 1–18, 2007.

35 Alexander A. Razborov. Lower bounds on the size of constant-depth networks over a complete basis with logical addition. *Mathematicheskie Zametki*, 41(4):598–607, 1987.

36 Alex Scott, Peter Jeavons, and Lei Xu. Feedback from nature: an optimal distributed algorithm for maximal independent set selection. In *Symposium on Principles of Distributed Computing (PODC)*, pages 147–156, 2013.

37 Ronen Shaltiel and Emanuele Viola. Hardness amplification proofs require majority. In *Symposium on Theory of Computing (STOC)*, pages 589–598, 2008.

38 Roman Smolensky. On Representations by Low-Degree Polynomials. In *Symposium on Foundations of Computer Science (FOCS)*, pages 130–138, 1993.

39 John P. Steinberger. The Distinguishability of Product Distributions by Read-Once Branching Programs. In *Conference on Computational Complexity (CCC)*, pages 248–254, 2013.

40 Andrew Chi-Chih Yao. Separating the Polynomial-Time Hierarchy by Oracles. In *Symposium on Foundations of Computer Science (STOC)*, pages 1–10, 1985.

41 Zhiqiang Zhang and Yaoyun Shi. On the parity complexity measures of Boolean functions. *Theor. Comput. Sci.*, 411(26-28):2612–2618, 2010.

Space-Optimal Naming in Population Protocols

Janna Burman[1]
LRI, Université Paris-Sud, CNRS, Université Paris-Saclay, France
janna.burman@lri.fr

Joffroy Beauquier
LRI, Université Paris-Sud, CNRS, Université Paris-Saclay, France
joffroy.beauquier@lri.fr

Devan Sohier
LI-PaRAD, Université de Versailles, Université Paris-Saclay, France
devan.sohier@uvsq.fr

Abstract

The distributed *naming problem*, assigning unique names to the nodes in a distributed system, is a fundamental task. This problem is nontrivial, especially when the amount of memory available for the task is low, and when requirements for fault-tolerance are added.

The considered distributed communication model is *population protocols*. In this model, a priori anonymous and indistinguishable mobile nodes (called agents), communicate in pairs and in an asynchronous manner (according to a fairness condition). Fault-tolerance is addressed through *self-stabilization*, in terms of arbitrary initialization of agents.

This work comprises a comprehensive study of the necessary and sufficient state space conditions for naming. The problem is studied under various combinations of model assumptions: *weak or global fairness*, arbitrary or uniform initialization of agents, existence or absence of a distinguishable agent (arbitrarily initialized or not), possibility of breaking symmetry in pair-wise interactions (*symmetric or asymmetric transitions*). For each possible combination of these assumptions, either an impossibility is proven or the necessary *exact* number of states (per mobile agent) is determined and an appropriate space-optimal naming protocol is presented.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases networks of passively mobile agents, population protocols, deterministic naming, self-stabilization, exact space complexity, tight lower bounds, global and weak fairness

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.9

Related Version A full version of the paper is available at [8], https://hal.inria.fr/hal-01790517.

1 Introduction

The population protocol model was introduced as a minimalist model for mobile sensor networks where the computational devices, called agents, communicate by pair-wise interactions without (almost) any control on their communication schedule [1].[2] The model originally assumed that the memory of agents is constant, i.e., independent of the population size n. Due to this, agents are prohibited from storing unique identifiers.

Many limitations have been discovered due to this restriction, and a lot of work has been devoted to study the power added by relaxing it. For example, it was obtained in [10] that a non-semi-linear predicate can be computed starting from $\Theta(\log\log n)$ bits of agent's memory (allowing $\Theta(\log^{O(1)} n)$ identifiers), while the original population protocols compute only exactly the semi-linear predicates [2]. Furthermore, it was shown that using $\Theta(\log n)$

[1] corresponding author
[2] Basically, at each step a pair of agents is scheduled to interact (subject to a fairness condition) and each agent observes the other's state updating its own according to the transition function.

© Janna Burman, Joffroy Beauquier, and Devan Sohier;
licensed under Creative Commons License CC-BY

bits per agent, allowing to assign unique identifiers (names) to agents, already permits to compute exactly all the symmetric predicates in the class $NSPACE(n \log n)$ (what is equivalent to the power of $O(n \log n)$ space Turing machine) [15, 10]. Following these results, a comprehensive study in [7] provided a complete hierarchy establishing complexity classes for the cases going from non-identified agents (the original population protocol model) to uniquely identified ones, passing by the case of homonyms.

Another line of works where the possibility of having names plays an important role concerns fault-tolerant population protocols. As a motivating scenario one can think of reliability critical mobile sensor networks (not necessary of a very large scale), which may be hardly accessible and have to recover automatically to the correct behavior after faults. In the framework of *self-stabilization* [13, 3], i.e. when the reliability concerns transient faults (e.g., memory or communication errors), it was shown that a linear (on n) state space is necessary for the realization of many tasks. Interestingly, in these cases, many proposed solutions perform a sort of naming mechanism. This concerns for example the self-stabilizing tasks of leader election [9], counting [5, 16, 4] and deterministic oscillation [11]. In [12], for computing semi-linear predicates while tolerating a known constant number of transient and crash faults, the algorithm approximately divides the population into a predefined number of groups, actually creating named homonyms. Finally, some other fault-tolerant population protocols assume that names are given a priori, e.g., [15, 18].

These observations suggest that the task of naming in population protocols should receive a particular attention. This work presents a comprehensive study of this problem. It focuses on the necessary and sufficient state space conditions for naming under all possible combinations of a set of classical model assumptions, like existence of a leader, *weak* or *global* fairness, uniform or arbitrary initialization and symmetry of transition rules. Note that a choice of a combination affects the type and the level of difficulty of breaking symmetry or of achieving fault-tolerance. These parameters, their interest and effect are all discussed below.

The first parameter is the nature of the assumed fairness, weak or global. The formal definitions and an example illustrating the difference between the two appear in the model section. Intuitively, while global fairness ensures that an infinitely often reachable configuration is unavoidable, weak fairness only ensures that every pair of agents interacts infinitely often. Global fairness can be viewed as a way for modeling randomized systems (without introducing randomization explicitly in the model). This explains why it is generally easier to get solutions under this fairness. However, such randomization cannot be always assumed to be available, in particular in reliability critical systems.

A second parameter is the symmetry nature of the (transition) rules of a protocol. With *symmetric rules*, two interacting agents in the same state stay in identical states. With *asymmetric rules*, they can take different states. This latter assumption was the original one proposed for population protocols and motivated by asymmetric wireless communications. The symmetric rules assumption is more general (weaker), and many motivating scenarios can be found for it in nature inspired population protocols, social networks (when equity is an issue) or in networks with symmetric wireless communication.[3]

A third parameter is the presence or absence of a unique leader (a distinguishable agent),

[3] Note that an asymmetric population protocol can be transformed into a symmetric one using the transformer of [6]. However, this transformer requires global fairness and doubles the number of states per agent. This makes it frequently inadequate for obtaining a space efficient symmetric solution from an asymmetric one (in terms of exact space complexity), and certainly inadequate under weak fairness.

which is obviously also useful for the sake of breaking symmetry. In the context of sensor networks, this agent may represent a (possibly mobile) base station, having augmented resources comparing to the tiny mobile agents. From this perspective and similarly to previous studies (e.g., [20, 4, 16]), we are not concerned with the space complexity of this agent.

A fourth parameter is related to the initialization of system agents, i.e., of the leader (if present) and of the other agents (called here *mobile*). In case of mobile agents, if initialization is assumed, it is always *uniform*, i.e., to the same value for every mobile agent. In case of arbitrary initialization, the given solution stabilizes starting from an *arbitrary* configuration (i.e., resulting from any number of transient faults). In this case it is called self-stabilizing [13]. The weaker initialization assumption is (e.g., only the leader is initialized, or nobody), the stronger the system is against transient faults and the more adapted to repetitive execution of a task (requiring less or no re-initialization).

Finally, the focus of this work is on deterministic protocol design, useful whenever random behavior is inappropriate or deterministic guarantees are required. Moreover, as many previous works [20, 11, 4, 16, 5], we perform an *exact* state space analysis. On the negative side, this requires especially careful analysis and design, in contrast to, e.g., the asymptotic analysis case. On the positive side, the exact analysis is extremely relevant in the cases of particularly memory-limited devices, small sized networks and self-stabilizing protocols. The less volatile memory is used by a self-stabilizing protocol, the less (probabilistically less frequently) it is vulnerable to corruptions.

Contribution

Thus, we investigate the naming problem under all the possible combinations of the parameters described above. All the negative results (impossibilities and lower bounds) are presented in Section 3, while all the positive ones (state-optimal solutions) are given in Section 4. Table 1 gives a synthetic view of these results. For each case, it indicates first the statement establishing the feasibility, either by proving impossibility or by construction of a space-optimal protocol. In the latter case, the table also indicates the optimal (used) number of states and the relevant statement of the lower-bound.

The results are expressed in terms of P - a known upper bound on the number of mobile agents n (i.e., the maximum number of agents destined to be named). This technical assumption is done for dealing with bounded protocols (similarly to the related studies, e.g., [16, 5]). P can be seen as a function of the manufactured memory size in each (undistinguishable) mobile agent.[4] Nevertheless, the results (and the lower bounds in particular) can be equivalently expressed in terms of n, when considering a particular population size.

Notice that, first of all, the table concerns the case of *arbitrarily initialized mobile agents*. Together with that, it is also relevant to the case of *uniform initialization of mobile agents*. It is obvious for the positive results (the protocols stay correct). However, somewhat surprisingly, the impossibilities and lower bounds also hold under this stronger assumption, but with only one exception. The exception concerns symmetric rules under weak fairness and with an initialized leader, i.e., when a leader is present and can be initialized together with the other agents (refer to the table). Then, there is a simple naming protocol with only P states per

[4] The parameter P is used in the protocols, but this explicit usage can be frequently replaced by an alert mechanism announcing that the bound will be shortly reached.

mobile agent (Proposition 14), instead of the necessary $P+1$ states with arbitrary initialized mobile agents. In contrast with this simple protocol for completely initialized case, the analysis of tight negative and positive results assuming no initialization of mobile agents is much more complex (see Section 3.1 and Propositions 16 and 17).

Additional remarks can be made about the table. First, under weak fairness and without leader, no symmetric deterministic protocol is capable of breaking symmetry, and thus naming is impossible. Second, if asymmetric rules are allowed, in all cases, there is a space-optimal solution with P states. On the contrary, with symmetric rules, the norm is $P+1$ states, excluding two cases: when there is an initialized leader (in both cases) and (1) either global fairness or (2) uniform initialization of mobile agents is assumed. In both cases, the problem can be solved with P states per agent.

Table 1 Synthesis of the relevant statements establishing the feasibility of naming and the necessary (optimal) state space, under different model parameters. If not stated otherwise, the indicated results hold for both arbitrarily an uniformly initialized mobile agents.

	symmetric rules		**asymmetric rules**
	weak fairness	**global fairness**	**weak/global fairness**
no leader	impossible (Prop. 1)	Prop. 13, with $P+1$ states (Prop. 3)	Prop. 12, with P states
non-initialized leader	Prop. 16, with $P+1$ states (Prop. 4)	Prop. 13, with $P+1$ states (Prop. 4)	Prop. 12, with P states
initialized leader	*non-initialized agents:* Prop. 16, with $P+1$ states (Theorem 11); *initialized agents:* Prop. 14, with P states	Prop. 17, with P states	Prop. 12, with P states

Note. Due to the space constraints, several proofs and the additional related work discussion have been moved to [8]. Though, the most relevant work is mentioned above.

2 Model and Notations

We adopt the model of population protocols [1] as a basic model. A system consists of a collection $\mathcal{A}$ of pairwise interacting agents, also called *population*. Each agent can represent a sensing and communicating mobile device. Among the agents, there can be a unique distinguishable agent called the *leader* which can be as powerful as needed, in contrast with the resource-limited non-leader agents. The non-leader agents are also called *mobile*, interchangeably. The size of the population n is the number of the mobile agents. It is unknown (a priori) to the agents, contrary to the upper bound parameter $P \geq n$. Each agent has a state taken from a set of states Q (depending on P), the same for all mobile agents, but generally different for the leader.

A *(population) protocol* can be modeled as a finite transition system defined by transitions between *configurations*, where a configuration is a vector of states of all the agents. The transitions between configurations are defined by pairwise interactions between agents. If, in a configuration C, two agents x and y interact (meet), s.t. x is in state p and y in state q,

they execute a *transition rule* $(p, q) \to (p', q')$. As a result, x changes its state from p to p' and y from q to q'. The new configuration C', resulting from this state changes, is said to be reachable from C (in one step), denoted by $C \to C'$. If $p = p'$ and $q = q'$, the corresponding transition is said *null* (such transition rules are specified by default), and non-null otherwise.[5] If there is a sequence of configurations $C = C_0, C_1, \ldots, C_k = C'$, such that $C_i \to C_{i+1}$ for all $i, 0 \leq i < k$, we say that C' *is reachable* from C, denoted $C \xrightarrow{*} C'$.

The transition rules are *deterministic*, if for every pair of states (p, q), there is exactly one (p', q') such that $(p, q) \to (p', q')$. We consider only deterministic transitions and thus, only *deterministic protocols*. Transitions and protocols can be *symmetric* or *asymmetric*. Symmetric means that, if $(p, q) \to (p', q')$ is a transition rule, then $(q, p) \to (q', p')$ is also a transition rule. In particular, if $(p, p) \to (p', q')$ is symmetric, $p' = q'$. A protocol is called symmetric, if all its transition rules are symmetric, and asymmetric otherwise.

Let $(p_1, q_1) \to (p_2, q_2)$, $(p_2, q_2) \to (p_3, q_3)$, ..., $(p_{k-2}, q_{k-2}) \to (p_{k-1}, q_{k-1})$, $(p_{k-1}, q_{k-1}) \to (p_k, q_k)$ be a sequence of rules of a protocol. Then, we shortly write $(p_1, q_1) \xrightarrow{*} (p_k, q_k)$ to denote the successive application of the rules of the sequence, to the same pair of agents, initially in states p_1 and q_1, leading them to p_k and q_k, respectively. We sometimes call an agent in state p a p-state agent, or just a p-agent.

An *execution* of a protocol is an infinite sequence of configurations and transitions $C_0, t_1, C_1, t_2, C_2, t_3, \ldots$ such that C_0 is the starting configuration and for each $i \geq 0$, $C_i \to C_{i+1}$, t_i being the transition between two particular agents used to reach C_{i+1} from C_i. When the actual transitions are irrelevant, we denote the execution by $C_0, C_1, C_2, \ldots$. A *segment* or a *sub-execution* is a sub-sequence of an execution. The *trace of transitions* of a sub-execution $C_0, t_1, C_1, t_2, C_2, \ldots t_k, C_k$ is a sequence $t_1, t_2, t_3, \ldots t_k$ of transitions, and the corresponding *trace of transition rules* is the sequence $r_1, r_2, r_3, \ldots r_k$ such that r_i is the transition rule applied in t_i. In a real distributed execution, interactions of distinct agents are independent and could take place simultaneously (in parallel), but when writing down an execution we order those simultaneous interactions arbitrarily.

An execution is said *weakly fair*, if every pair of agents in $\mathcal{A}$ interacts infinitely often. An execution is said *globally fair*, if for every two configurations C and C' such that $C \to C'$, if C occurs infinitely often in the execution, then C' also occurs infinitely often in the execution. This also implies that, if in an execution there is an infinitely often reachable configuration, then it is infinitely often reached [2]. Note that an execution where pairs of agents interact according to some probabilistic distribution is globally fair with probability 1 [17].

A simple example allows to better understand the difference between weak and global (or probabilistic) fairness. Consider a population of 3 agents. Each agent can be white or black, and initially one agent is black and the two others are white. Consider also the protocol in which, when two white agents interact, they both become black and when two agents of different colors interact, they exchange their colors. It is easy to see that there is an infinite weakly fair execution in which there is always one black and two white agents (the black color "jump" indefinitely from agent to agent). On the contrary, every globally fair execution terminates in a configuration in which the 3 agents are black, because otherwise there would be infinitely many configurations during an execution from which the "all black" configuration could be reached, without ever being reached (contradicting global fairness).

A *(static) problem* is defined by a predicate $\mathcal{D}$ on configurations. A population protocol

[5] For simplicity, in some cases, we do not present protocols under the form of transition rules, but under the equivalent form of a pseudo-code.

$\mathcal{P}$ is said to *solve a problem* $\mathcal{D}$, if and only if every execution of $\mathcal{P}$ reaches a configuration satisfying the conditions defining $\mathcal{D}$ and stays in such configurations forever after. When this happens, we say that the protocol has *stabilized* (or *terminated*), and a *terminal configuration* has been reached. A *self-stabilizing protocol* is a protocol that stabilizes from an arbitrary configuration (i.e., from a configuration where *any* agent, *including the leader*, can be in any possible state).

In the *naming* problem, each mobile agent x has a variable, also called a *name*, that eventually does not change and such that no two agents have the same name. Mobile agents having the same state (thus the same name) are called *homonyms*.

We consider *uniform* or *semi-uniform* protocols (cf. [14, 19]) in the sense that all agents, except the leader (whence semi-), are a priori indistinguishable and interact according to the same transition rules. Moreover, given an upper bound P on n, the protocol functions similarly for any n. Thus, given the bound P, by the definition of naming, an obvious lower bound on the state space of a mobile agent (for solving naming) is P.

3 Negative Results

In this section, we clarify some boundaries on the possibility to obtain symmetric naming. They are summarized in Table 1 and useful for establishing the space-optimality (tightness) of the solutions presented in the next section. We proceed from simple to more intricate results, gradually adding assumptions helping in breaking symmetry (making harder to proof impossibilities). We conclude this section by Theorem 11 - a subtle result stating impossibility to get a P state symmetric protocol even with an initialized leader, but non-initialized mobile agents and under weak fairness.

The first result is obtained by observing a completely symmetric weakly fair execution where at each step the population transits from one uniform configuration (all agents are in the same state) to the other, by applying each time a symmetric rule between two homonyms.

▶ **Proposition 1.** *Under weak fairness and without leader (even with a uniform initialization of mobile agents), symmetric naming is impossible in the population protocol model.*

Proof. By contradiction, assume that such a symmetric protocol $\mathcal{PP}$ exists. With or without an initialization, consider a possible starting configuration where each agent is in state s_1 and the population size is even (this also corresponds to a uniform initialization). We build a weakly fair infinite execution of $\mathcal{PP}$ during which no configuration with agents in distinct states is reached. This execution can be described by phases. In the first phase, the agents are matched in pairs and interact accordingly. As the protocol is symmetric, after this first phase, each agent is in some state s_2, the same for all agents. In the next phase, the agents are matched again in pairs, but differently from the previous phase, and interact accordingly. After this phase, each agent is in some state s_3, the same for all agents. The execution continues in such phases such that eventually every agent has interacted with every other. From now on, such interaction sequence is repeated infinitely often, satisfying weak fairness. However, this execution never assigns distinct names to agents. ◀

The following lemma states properties of the transition rules of any P state symmetric naming protocol. Its short proof is given below. The lemma is used in the proofs of the next two propositions: the first one assumes no existing leader, while the second one proves impossibility of a self-stabilizing symmetric naming even with a leader.

▶ **Lemma 2.** *In any symmetric naming protocol using only P states per agent, the only possible non-null transition rules between two non-leader agents are between two homonyms.*

Proof. Otherwise, in a population of P agents, after stabilization (every state in $\{1, \ldots, P\}$ is assigned to some agent), mobile agents would be able to change their states and hence their names. This is a contradiction to the assumed stabilization. ◀

Under the conditions of the next proposition and by the lemma above, the only non-null transitions are between two homonyms (resulting in another two homonyms). Such transitions are useless for eliminating repeated names, implying the result.

▶ **Proposition 3.** *Even with a uniform initialization of agents, but without a leader, and using only P states per agent, it is impossible to obtain symmetric naming in the population protocol model, under weak or global fairness.*

▶ **Proposition 4.** *There is no symmetric naming protocol with P states per agent with an arbitrarily initialized leader (or without it), under weak or global fairness.*

Proof. Assume by contradiction that such a protocol exists. By Proposition 3, a leader is necessary. Consider a population of P mobile agents and a starting configuration C_0 (with possibly uniformly initialized mobile agents) that has homonyms (with states in $\{1, \ldots, P\}$). By Lemma 2, only transitions with a leader can eliminate homonyms (the only possible non-null symmetric transition rules between homonyms create necessarily two other homonyms). Thus, there is a finite execution sequence e, starting from C_0, during which the leader renames interacting mobile agents, and finally stabilizes to a correct naming (where every name from $\{1, \ldots, P\}$ is assigned to some mobile agent). Denote by C_e and by s_e the configuration and the state of the leader, respectively, at the end of e.

Then, assume a starting configuration C'_0 (with possibly uniformly initialized mobile agents, as before) containing only homonyms in state s (in $\{1, \ldots, P\}$) and with the leader in state s_e. There must be a sequence of interactions between the leader and mobile agents starting from C'_0 which ends up with a transition during which an interacting agent changes its name. Such a sequence of interactions exists also starting from C_e (with either weak or global fairness), because in C_e any possible state exists in the population. This contradicts the assumption that the protocol has stabilized starting from C_e in e. Hence, the proposition follows. ◀

3.1 Impossibility of P state symmetric naming of arbitrarily initialized mobile agents, under weak fairness, even with an initialized leader

For proving this stronger impossibility result, let us assume, for the sake of contradiction, that such a solution exists. Thus, denote by $Name$ any symmetric protocol solving the naming problem under weak fairness (for any $n \le P$), with arbitrarily initialized P-state mobile agents. By Proposition 3, under such conditions, a leader is necessary. Moreover, by Proposition 4, such an agent has to be initialized. So, in this sub-section, we assume such initialized unique leader. In the following, some additional necessary properties of protocol $Name$ are proven and finally imply the impossibility of its existence (see Theorem 11).

Using the lemma below, the next proposition establishes an important property of any protocol $Name$ - the existence of a unique state m, called *sink*. This particular state satisfies the following three conditions: (1) $(m, m) \rightarrow (m, m)$; (2) for every state $s \in Q$, there is a transition rule sequence $(s, s) \xrightarrow{*} (m, m)$; (3) for any $n < P$, no mobile agent is assigned a name m at stabilization (i.e., m does not appear infinitely often in executions with $n < P$).

▶ **Lemma 5.** *Consider any weakly fair execution $e = C_1, C_2, C_3, \ldots, C_j, \ldots$ of $Name$ on a population $\mathcal{A}$ of size $n < P$. There is an integer k such that, for any $j \ge k$, no mobile agent is in a state $m \in Q$ such that there is a sequence of transitions of $Name$ $(m, m) \xrightarrow{*} (m, m)$.*

Proof. Let us assume, by contradiction, that there are infinitely many configurations in e with a mobile agent in state m. Since there is a finite number of agents, there is a particular mobile agent x in $\mathcal{A}$ which is in state m in infinitely many configurations. Let $C_{j_1}, C_{j_2}, C_{j_3}, \ldots$ be these configurations such that $e = e_1, C_{j_1}, e_2, C_{j_2}, e_3, C_{j_3}, \ldots$ W.l.o.g., we choose these configurations such that, in every execution segment e_i, every agent in $\mathcal{A}$ interacts with every other (this is possible with weak fairness).

Now consider a population $\mathcal{A}' = \mathcal{A} \cup \{x'\}$ of size $n+1$. To prove the lemma, we will construct a weakly fair execution e' of $Name$ in population $\mathcal{A}'$ where no agent can distinguish e' from e, and where consequently $Name$ wrongly names the agents. Precisely, in e', x and x' will be simultaneously in state m in infinitely many configurations.

We construct e' based on e. First, we assume that in e', x' is in state m in the starting configuration, and $e' = e'_1, C'_{j_1}, e^m, e'_2, C'_{j_2}, e^m, e'_3, C'_{j_3}, e^m, \ldots$ Every segment e'_i follows exactly the same transition sequence as in e_i. In every segment $e'_{2r+1}, C'_{j_{2r+1}}$ (for $r \geq 0$) the interactions are exactly the same as in $e_{2r+1}, C_{j_{2r+1}}$, and x' does not interact. However, in $e'_{2r}, C'_{j_{2r}}$, all the interactions are as in $e_{2r+2}, C_{j_{2r}}$, but the interactions with x. In this case, x is replaced by x' in the appropriate state, and x does not interact. Finally, e^m is an execution segment where only x and x' interact. They both start in state m, performing the sequence $(m, m) \xrightarrow{*} (m, m)$. The configurations at the beginning and at the end of e^m are identical. The construction of e' ensures that in every C'_{j_i}, both x and x' are in the state m.

It is easy to verify that e' is possible. In particular, this is because, at the end of every segment e'_i, C'_{j_i}, e^m, both x and x' are in the state m, so they can be exchanged in the following transitions of e'_{i+1}. Moreover, e' is weakly fair, because x' interacts with x in every e^m; in every e'_{2r+1} and e'_{2r+2}, x and x', respectively, interact with every other agent (by the assumption on e_i); and all the other agents interact with all the others infinitely often, by the later arguments and by weak fairness of e.

Finally, in e', $Name$ does not name x and x' differently. This is a contradiction to the assumption that $Name$ is a correct naming protocol. ◀

▶ **Proposition 6.** *In any protocol $Name$, there is a sink state.*

Proof. As $Name$ is symmetric, two interacting agents, both in some state $s \in Q$, execute some symmetric transition of the form $(s, s) \to (s_1, s_1)$. If they meet several times successively, there is a possible sequence of transitions $(s, s) \to (s_1, s_1) \to (s_2, s_2) \to (s_3, s_3) \ldots$ As mobile agents are finite state, for some $j > i \geq 1$, $s_i = s_j$, i.e. $(s_i, s_i) \xrightarrow{*} (s_i, s_i)$. By Lem. 5, $s_i = m$ s.t. m does not appear infinitely often in executions with $n < P$. As there are at least $P-1$ states appearing infinitely often in an execution with $n = P-1$, there is at most one such possible state m in a P state protocol. This implies correctness of conditions (2) and (3) of the sink definition.

Finally, by contradiction, if $(m, m) \to (s, s)$ s.t. $s \neq m$, then the previous part of the proof implies $(s, s) \xrightarrow{*} (s, s)$. As m is proved (above) to be unique, this is a contradiction, implying the correctness of condition (1) of the sink definition. ◀

From now on, the proof assumes the sink state denoted by m, and shows the impossibility for a particular kind of executions, called here *reduced.* In any segment of a reduced execution, each time a pair of $s \neq m$ homonyms appears, it is immediately *reduced* to m, i.e., by applying the sequence of transition rules $(s, s) \xrightarrow{*} (m, m)$, whose transitions and by extension the sequence itself are called *(homonym) reducing.* Thus, other transitions can take place only when there are no more homonyms. Naturally, any configuration without any homonyms except those in state m is called *reduced.* Notice that, in a reduced execution, there are non-reduced configurations, but only during the reducing transition sequences. For

example, consider the following reduced sub-execution (where l_i represents a leader state): $[1,2,3,4,m,l_1],(l_1,1)\to(l_2,2),[2,2,3,4,m,l_2],(2,2)\to(m,m),[m,m,3,4,m,l_2],(l_2,m)\to(l_3,3),[m,3,3,4,m,l_3],(3,3)\to(m,m),[m,m,m,4,m,l_3]$. In this example, the reduced configurations are $[1,2,3,4,m,l_1]$,
$[m,m,3,4,m,l_2]$ and $[m,m,m,4,m,l_3]$.

Forcing a reducing sequence of transitions whenever possible, as in a reduced (sub-) execution, does not prevent an execution from being weakly fair. Thus, we can prove the following corollary.

▶ **Corollary 7.** *Given protocol $Name$, any reduced sub-execution of $Name$ can be prolonged to a reduced weakly fair execution of $Name$, i.e., in which $Name$ stabilizes.*

Proof. First, recall that, by Prop. 6, and Lemma 2, the only non-null-transitions of $Name$ are the homonym reducing transitions between non-m-mobile agents and the transitions with the leader. Given a reduced segment, we prolong the execution by forcing mobile agents to interact with every other agent (including the leader) in a "round-robin fashion" (not necessarily in consecutive interactions). Whenever homonyms are created, this interaction pattern is interrupted by the homonym reducing sequence of transitions, and then resumed after the reduction. The execution constructed that way is weakly fair and uses only transitions of $Name$, hence it stabilizes towards a naming. ◀

The following lemma states a basic property of $Name$, in a population of P agents. It uses the notion of equivalent configurations. Two configurations are *equivalent* if both correspond to the same multi-set of states[6] (e.g., $C_1=[2,3,2,m,l]$ is equivalent to $C_2=[2,2,3,m,l]$, where l stands for the leader state). This notion is naturally extended to equivalent executions. The lemma shows that, in a population of P agents, if there is a reduced sub-execution in which some particular state $s\neq m$ never appears (in its reduced configurations), than there is also an *equivalent* sub-execution where a particular m-agent never interacts.

▶ **Lemma 8.** *In a population of P agents, consider a reduced sub-execution $e=CC_1C_2\dots C_k$ of $Name$, starting from a reduced configuration C and such that no agent in state $s\neq m$ (for some s) exists in any reduced configuration of e. Then, there exists a reduced sub-execution $e'=CC'_1C'_2\dots C'_k$ of $Name$ in which a particular m-agent x never interacts, and, as in e, no agent in state $s\neq m$ exists in any reduced configuration of e'. Moreover, every configuration C'_i of e' is equivalent to the configuration C_i in e.*

Proof. In a population of P agents, in any reduced configuration (of a reduced execution of $Name$), there is at least one m-state agent. Thus in any reduced configuration of e there are at least two m-state agents. Consider C and call one of these two agents agent x.

Let us construct now e', starting in C, with exactly the same trace of transition rules of e. By Prop. 6, and Lemma 2, the only non-null-transitions are homonym reducing transitions between non-m-mobile agents and the transitions with the leader. By a simple induction below, one can see that every transition rule in a trace of e, step by step, can be executed without participation of agent x, and thus added to e'. Since no transition of e creates an s-agent, no such added transition can create an s-agent.

Thus, starting in C, the first transition of e can be executed with any agent, except x, and thus can be added to e (the base of induction). The resulting configuration C'_1 is

[6] or if one is a permutation of the vector components of the other (recall that a configuration is a vector of states of the agents)

equivalent to C_1. Then, by induction, assume that after k transitions added to e', the reached configuration is equivalent to the one reached after transition k in e (without any s-state agent in a reduced configuration). For $k+1$, if a homonym reducing transition takes place in e, x does not participate (it is already reduced) and thus the same transition can be added to e', and an equivalent configuration is reached. Otherwise, if this is a reduced configuration, there is an additional m-state agent, different from x, so any transition with the leader can be executed excluding x, and an equivalent configuration is reached. By such construction of e', every configuration C'_i is equivalent to C_i in e. ◀

Now, we want to prove that $Name$, in a population of size P, can reach a terminal configuration C (with uniquely named agents and in particular with an agent x in some state $s \neq m$), but such that the leader may be unaware of that. Then, the leader should manage to create the possibly missing s-agent for stabilizing towards a configuration where naming is realized. But, once created, this s-agent can disappear by a reduction with the "hidden" s-agent x. This contradicts the fact that the reached configuration C is terminal (see the proof of Theorem 11). This proof uses the following two technical lemmas.

The first lemma (Lem. 9) states that, in some conditions, from a reduced configuration with an s-agent, a reduced configuration without such an agent is reachable. The second symmetric lemma (Lem. 10) states that, if there is some constrained sub-execution to a latter configuration without an s-agent, then there also exists a particular sub-execution from the configuration without an s-agent to the one with it. We use the following definitions to describe these aspects formally.

A configuration C_1 is said to be *far away* from C_2 *by one state* $s \neq m$ (in agent x), if there is an agent x such that $C_1[x] = m$, $C_2[x] = s \neq m$ and $\forall y \in \mathcal{A} \setminus \{x\}, C_1[y] = C_2[y] \neq s$. Then, C_1 is denoted by C_2^{-s} and C_2 by C_1^{+s}. Agent x is called the *pivot*. For example, $C_1 = [1, m, 3, 4, m, l_1]$ is far away by one state 2 from $C_2 = [1, 2, 3, 4, m, l_1]$, and $C_1 = C_2^{-2}, C_2 = C_1^{+2}$.

▶ **Lemma 9.** *Consider a population of size P and two reduced configurations C_1 and C_1^{-s}, far away by state s, in agent x. Consider a given reduced sub-execution $C_1^{-s} e_1^{-s} C_2$ of $Name$ where: (i) there is no s-agent in every reduced configuration in the segment $C_1^{-s} e_1^{-s}$, (ii) agent x does not interact in $C_1^{-s} e_1^{-s} C_2$, (iii) C_2 is reduced and has exactly one s-agent. Then, there exists a reduced sub-execution $C_1 e_1 C_2^{-s}$ of $Name$ such that exactly one agent in state s exists in every reduced configuration in $C_1 e_1$, and agent x does not interact in C_1 e_1 C_2^{-s}, except in the very last (s-homonym) reducing sequence.*

Proof. Given a sub-execution $C_1^{-s} e_1^{-s} C_2$, the sub-execution $C_1 e_1$ is constructed, starting from a configuration C_1, by applying exactly the trace of transition rules of $C_1^{-s} e_1^{-s} C_2$, on the population excluding the s-state agent x (recall that agent x does not interact in $C_1^{-s} e_1^{-s} C_2$). At the end of the execution constructed till now, there are exactly two s-state homonyms (x and another s-agent). Now, the transitions reducing these homonyms to m are added to reach the desired configuration C_2^{-s}. In this way, the sub-execution $C_1 e_1 C_2^{-s}$ is obtained. Notice that agent x interacts only in the very last (s-homonym) reducing sequence. ◀

▶ **Lemma 10.** *Consider a population of size P and two reduced configurations C_1 and C_1^{-s}, far away by state s, in agent x. Consider a given reduced sub-execution $C_1 e_1 C_2^{-s}$ of $Name$ where exactly one agent in state s exists in every reduced configuration in the segment $C_1 e_1$, and agent x does not interact in C_1 e_1 C_2^{-s}, except in the very last (s-homonym) reducing sequence. Then, there exists a reduced execution $C_1^{-s} e_1^{-s} C_2$ of $Name$ such that there is no s-agent in any reduced configuration in the segment $C_1^{-s} e_1^{-s}$, and C_2 is reduced and has exactly one s-agent.*

Proof. In a given execution $C_1 e_1 C_2^{-s}$ agent x does not interact, except in the very last (s-homonym) reducing sequence. Starting from C_1^{-s} we construct an execution $C_1^{-s} e_1^{-s} C_2$, by using first exactly the same prefix of the trace of transition rules of $C_1 e_1 C_2^{-s}$, until and excluding the very last (s-homonym) reducing sequence. In the given configuration, before this very last reducing sequence, two s-agents necessarily exist, one of them being till now the non-interacting agent x. However, in the sub-execution constructed at this point, x does not interact either, but is in state m, so exactly one s-agent exists at the end of the constructed sub-execution. Hence, C_2 is reached and the required $C_1^{-s} e_1^{-s} C_2$ is obtained. ◀

▶ **Theorem 11.** *Under weak fairness, without the initialization of mobile agents, there is no symmetric naming protocol with P states per agent.*

Proof. By contradiction, assume that such a protocol $Name$ exists. Consider a population of P agents. Assume an agent x that does not communicate (for long enough), while the protocol is stabilizing with only $P-1$ mobile agents. By Proposition 6, when this happens, no agent, except possibly x, is in state m. Thus, consider two possible configurations. In one, C_1, the state of x is m, and in another, x is in state $s \neq m$. In the latter case, reduce the s-state homonyms (to m) (possible by Prop. 6). The reached configuration is C_1^{-s}.

Assume that the actual obtained configuration is C_1^{-s}. By the correctness of $Name$, starting from C_1^{-s}, any execution e stabilizes to a configuration C_* where all agents are in different states and no state changes thereafter. Moreover, by Corollary 7, there is such an execution, which is reduced. Furthermore, any such e can be decomposed s.t.
$e = C_1^{-s} e_1^{-s} C_2 \; e_2 \; C_3^{-s} e_3^{-s} C_4 \; e_4 \; C_5^{-s} \ldots C_k^{-s} e_k^{-s} C_* C_* \ldots$. For every odd i, no s-agent exists in any reduced configuration of $C_i^{-s} e_i^{-s}$, and C_i^{-s} is reduced. For every even i, exactly one s-state agent exists in every reduced configuration in $C_i e_i$, and C_i is reduced. By Lemma 8, for every odd i, one can choose a segment $C_i^{-s} e_i^{-s}$ such that a particular m-agent x_i never interacts in this segment.

Furthermore, e is chosen such that, for every segment $C_i e_i C_{i+1}^{-s}$ with an even i, a particular s-state agent $y_i \neq x_i$ does not interact, except in the very last (s-homonym) reducing sequence. Let us show (by induction) that such e exists. First, since in $C_i^{-s} e_i^{-s}$, for odd i, there is an m-agent $y \neq x_i$ in every reduced configuration. Thus, in the following C_{i+1} (even $i+1$) configuration, any such agent or other non-m-agent $y_i \neq x_i$ can become an s-agent. This implies that every agent in e has the opportunity to interact repeatedly. Second, by Lemma 9, $C_1 e_1 C_2^{-s}$ exists (such that there is exactly one agent in state s in every reduced configuration of $C_1 e_1$).Thus, starting from C_1, at the end of $C_1 e_1 C_2^{-s}$, no s-agent would exist. Then, by correctness of $Name$, there exists $C_2^{-s} e_2^{-s} C_3$ (where no s-agent exists in $C_2^{-s} e_2^{-s}$), and by Lemma 8, such that, in $C_2^{-s} e_2^{-s}$, a particular m-state agent does not interact, e.g., the pivot agent of C_2^{-s} and C_2. Thus, by Lemma 9, there exists e, such that, in $C_2 e_2 C_3^{-s}$, a particular s-state agent y_2 does not interact, except in the very last (s-homonym) reducing sequence (this proves the base of induction).

Now, one can repeat the same arguments, for any segment $C_i^{-s} e_i^{-s} C_{i+1} e_{i+1} C_{i+2}^{-s}$, for an odd i, in e, and show (by induction) that the chosen e exists. Recall that we assumed that $Name$ has stabilized in C^* and then, the leader has to stop renaming the agents. In addition, C_* is reduced (all agents are distinctly named). Hence, from this point, only null-transitions are possible.

Notice that the conditions of Lemma 9 are satisfied for the segments $C_i^{-s} e_i^{-s} C_{i+1}$ with an odd i, and the conditions of Lemma 10 are satisfied for the segments $C_i e_i C_{i+1}^{-s} e_i^{-s}$ with an even i. Hence, by applying these lemmas repeatedly to the segments of e, one inductively

builds the execution segment $e' = C_1 e_1 C_2^{-s} e_2^{-s} C_3 e_3 \dots C_k e_k C_*^{-s}$. However, C_*^{-s} is reduced, and far away by only one state from C_*. No agent except the pivot, can distinguish C_*^{-s} from C_*, since each one is in the same state in both configurations. In particular, $C_*^{-s}[leader] = C_*[leader]$. Thus, and by Lemma 2 and Prop. 6, the only possible transitions with the leader are null transitions, as well as the transitions involving mobile agents (there are no non-m-homonyms). Obviously, in C_*^{-s}, the protocol has not stabilized yet. But, no transition can change the configuration C_*^{-s}. This contradicts the assumption that $Name$ is correct. ◀

4 Positive Results

We start by a proposition that illustrates the power of asymmetric transition rules, compared to symmetric ones. Basically, with asymmetric rules, a leader is not necessary for breaking symmetry, even under weak fairness. Moreover, P states are sufficient and no initialization is necessary, i.e., self-stabilizing space-optimal naming is possible.

The proof is by construction of a protocol with a single asymmetric type of rule, $(s, s) \to (s, (s+1) \mod P)$. This idea is known in the literature, e.g., [9, 5]. It aimed at solving other (than naming) problems, but provided naming as a by-product. [9] considers self-stabilizing leader election, assuming that the exact size of the population n is known (an assumption proven to be necessary). Under this assumption, the presented protocol also solves naming. In [5], a similar idea is used to count the arbitrarily initialized mobile agents, assuming an initialized leader, and realizes also naming.

The asymmetric space-optimal naming protocol presented below is proven under more general assumptions (with upper bound P, instead of exact knowledge of n, without a leader, and under both fairness assumptions). Its proof uses the novel technique of *hole* and *hole distance* in a configuration.

▶ **Proposition 12.** *Even if agents cannot be initialized, asymmetric naming (under global or weak fairness) is possible using an optimal number of states (P) per agent and without leader.*

Proof. Consider the following asymmetric protocol with P-state agents and only one type of transition rules: $(s, s) \to (s, (s+1) \mod P)$.

To prove its correctness let us use the following definitions. A *hole in a configuration C* is an integer i such that no agent is in state i in C. The *hole distance of an agent, in state i, in a configuration C*, is the minimum positive integer j such that $i + j \mod P$ is a hole, if such an integer j exists, and 0 otherwise. The *hole distance of a configuration C* is the sum of the hole distances of the agents in C. Let f be the function mapping each configuration C to a pair of integers (number of holes in C, hole distance of C).

Let C and C' be two different configurations such that $C \to C'$. Let us show that, for the lexicographical order, $f(C) > f(C')$. First, remark that C' cannot have more holes than C. If C' has one hole less than C, we are done. If not (C' has the same holes as C), there is an agent in state i in C that has changed its state to $i+1 \mod P$ in C'. Thus, the hole distance of C' is the hole distance of C minus 1. We are done also in this case.

Since f is upper bounded (e.g., by $(P, P(P-1))$), there is a sequence of transitions that reaches a configuration from which only null transitions are possible, making no effect on agents' states, that are thus necessarily distinct. Then, naming is achieved. ◀

Now we consider one of the most difficult cases for symmetric protocols – assuming no leader and no initialization - impossible under weak fairness. Recall that global fairness mimics, in some sense, the behavior of randomized environments. The following proposition shows that this pseudo randomization is sufficient for breaking symmetry, and that a

distinguishable agent is not needed for that. Thus, we propose below the first *symmetric space-optimal self-stabilizing* naming obtained without a leader (the complete proof is in [8]). Notice, that by Proposition 3, at least $P+1$ states per agent have to be used in this case.

▶ **Proposition 13.** *Even if agents cannot be initialized and without a leader, symmetric (self-stabilizing) naming under global fairness, for $n > 2$, is possible using $P+1$ states per agent.*

Proof Sketch. Consider the following symmetric protocol with state space $Q = \{0, 1, \ldots, P\}$ and defined by three types of transition rules:
1. *if* $s \neq P : (s, P) \to (s, (s{+}1) \mod P)$; **2.** *if* $s \neq P : (s, s) \to (P, P)$; **3.** $(P, P) \to (1, 1)$.

From a configuration with homonyms, rule 2 can be applied repeatedly to obtain a configuration C' where there are only P-state homonyms, and possibly some other uniquely named agents. If, in C', no uniquely named agents exist, let us force transitions using rules 3 and then 1, to create at least one uniquely named agent. Then, rule 2 is applied again, to reach a configuration C with only P-state homonyms and with at least one uniquely named agent. Then, whenever there are still some P-state homonyms in C, pick an agent with a unique name s such that no agent with a unique name $(s+1) \mod P$ exists. Make the s-agent interact with some P-agent, applying rule 1. The obtained configuration contains one more unique name than in C. If naming is not yet reached, this scenario is repeated until it is reached. Such an execution segment is possible from any configuration with homonyms. Hence, naming is reached in any globally fair execution. ◀

Up to this point, we have covered the possible positive cases assuming that no leader is present. Now, we show that the impossibility results of Section 3 can be circumvented by the assumption of a distinguishable agent. This allows to obtain three space-optimal symmetric protocols: 1) a simple P state protocol with all agents being initialized, including the leader (Prop. 14; its proof is in [8]); and two more intricate protocols: 2) a self-stabilizing one (with $P+1$ states) under weak fairness (Prop. 16); and 3) a protocol using only P states under global fairness (Prop. 17).

▶ **Proposition 14.** *Given a unique initialized leader, and uniform initialization of mobile agents, symmetric naming is possible using only P states per agent, under weak or global fairness.*

The next two results (Prop. 16 and 17) exploit the existing space-optimal *counting* protocol from [4], which uses P states per mobile agent, and (deterministically and exactly) counts such non-initialized agents under weak fairness, assuming an initialized leader . Let us denote it by $Count_P$. It was not originally intended to be a naming protocol, neither a self-stabilizing one. However, it can be observed that, for the case of $n < P$, it performs (a non self-stabilizing) naming. This property is reflected in Theorem 15 below.

▶ **Theorem 15** ([4]). *Protocol $Count_P$ in [4] solves the counting problem, under weak fairness, for up to P mobile agents, each with P states. Moreover, for any $n < P$, the protocol names (up to $P-1$) mobile agents with distinct names in $\{1, \ldots n\}$.*

By augmenting the mobile agents' state space to $P+1$ and adapting $Count_P$ accordingly, one obtains a naming protocol (also correct in the case where $n = P$), though using a non-optimal $P+1$ number of states. Let us denote the resulting protocol by $Count_{P+1}$. This protocol is adapted here further for solving the naming problem in a self-stabilizing way – Prop. 16, while the protocol of Prop. 17 is based on the original $Count_P$.

For presenting these protocols, some more details of *Count* have to be given. First, note that, in the new protocols here, an appropriate *Count* protocol is a priori executed independently, by every agent. The leader (also assumed in *Count*) manages an estimate for the population size. Let us denote it here by $Count.N$ (or just N when the particular version of *Count* is clear from the context). In the original version of *Count*, N is initialized to 0 and incremented until reaching the actual size n (N is non-decreasing). Each mobile agent x in *Count* has an arbitrary initialized variable $name_x$, which eventually contains a unique name (in $\{1, \ldots, n\}$), with $Count_{P+1}$ (for any $n \leq P$), or with $Count_P$ for any $n < P$. Only the leader can assign a new (non 0) name to an interacting agent in state 0. This state plays the role of the sink state (see definitions in Sect. 3.1). The only action of mobile agents is to reduce homonym states to the sink. Because of that, 0-agents appear along an execution until naming with names in $\{1, \ldots, n\}$ is reached. Notice that, even though naming may not be reached in $Count_P$ (0-agents may persist forever), it terminates, i.e., agents will eventually execute only null-transitions. This happens whenever a reduced configuration satisfying $Count_P.N = n$ is reached (this is used in Protocol 1, Prop. 17).

▶ **Proposition 16.** *Self-stabilizing (every agent state is initialized arbitrary) symmetric naming under weak fairness is possible using $P + 1$ states per mobile agent, given a unique (non-initialized) leader.*

Proof. By Theorem 15, for any $n \leq P$, $Count_{P+1}$ assigns unique names in $\{1, \ldots, n\}$ to mobile agents, if the leader is well initialized. However, to get a self-stabilizing version, one have to abandon this latter assumption (the leader cannot be initialized). Then, it may happen that $Count_{P+1}.N$ starts in a non 0 value and reaches $P + 1$, before the naming (and the correct count) is realized. Notice that in this case, mobile agents in state 0 exist (since a naming is not yet reached).

To overcome this case, we incorporate a reset technique. When a mobile agent x in state 0 ($name_x = 0$) interacts with the leader and the estimate $Count_{P+1}.N$ is bigger than P, the leader resets its internal variables (together with N) to the initialization values of the original $Count_{P+1}$. It is clear that, whenever such a reset is executed, the required naming is eventually and correctly obtained, by the correctness of $Count_{P+1}$. This solves the only problematic case described above. Hence, the proposition follows. ◀

Now, for proving Prop. 17, a protocol using only P states per mobile (non-initialized) agent is constructed. It is done by modifying protocol $Count_P$ and by exploiting the global fairness assumption. The resulting protocol is not self-stabilizing, as the leader is initialized (and otherwise impossible by Prop. 4 - with only P states). Notice that this case is similar to the conditions of Theorem 11 (stating impossibility of naming) except for the fairness condition. In fact, it is not easy to see why a similar reasoning used to prove Th. 11 (roughly the fact that the leader can never detect whether the naming is achieved or not) does not also hold in the current case (with global fairness). Intuitively, this is because the power of global fairness allows to eventually reach a favorable (for stabilization) sub-execution even using a deterministic protocol (though a particular one), while with weak fairness unfavorable sub-executions may persist forever.

▶ **Proposition 17.** *With an initialized leader (without initialization of mobile agents), symmetric naming under global fairness is possible using only P states per mobile agent.*

Proof. The proposed protocol is a modification of $Count_P$ in the code of the leader. The mobile agents are reducing homonyms to the sink state as in the original $Count_P$. The modified code is given below - Protocol 1. For every $n < P$, the new protocol works in

the same way as $Count_P$. Hence, by Theorem 15, it eventually stabilizes to naming for every $n < P$. The case of $n = P$ is treated separately, in lines 3 - 8. For this case, a variable *name_ptr* with possible values in $\{0, \ldots, P\}$ is used for indicating to the leader the name to assign to a mobile agent x interacting with it (using variable $name_x$, from the original protocol). The variable *name_ptr* is initialized to 0. Below we consider only the case of $n = P$.

Whenever $n = P$, the leader increments *name_ptr* each time it meets a mobile agent whose name is the current value of *name_ptr*. Otherwise, the agent is named by the value of *name_ptr*, and *name_ptr* is reset.

Let us consider only reduced (to 0) executions (see the definition in Sec. 3.1). From any *non-terminal* configuration, there is the following possible sequence of interactions during which the leader first resets *name_ptr* (if not 0 due to initialization), and then meets the existing $j < P$ uniquely named agents in the increasing order of their names $0, 1, 2, 3, \ldots j-1$. Variable *name_ptr* is then increased to j (≥ 1). After, the leader meets an agent in a state different from j. It names the agent by the current value of *name_ptr* ($= j$), and resets *name_ptr* again. Then, the scenario repeats, until *name_ptr* (and j) reaches P. No value can be changed thereafter, and all agents are named by names in $\{0, \ldots, P-1\}$. By global fairness, this terminal naming configuration is eventually reached. ◀

Algorithm 1 Space-Optimal Naming under Global Fairness (P states per mobile agent).

Variables at the leader:
  name_ptr: $[0, \ldots, P]$, initialized to 0
Variable at a mobile agent x**:**
  $name_x$: non-negative integer in $[0, \ldots, P-1]$, initialized *arbitrarily*

```
1:  when a mobile agent x interacts with the leader do
2:    execute Count_P
3:    if Count_P.N = P ∧ name_ptr < P then
4:      if name_x = name_ptr then
5:        name_ptr ← name_ptr + 1
6:      else
7:        name_x ← name_ptr
8:        name_ptr ← 0
9:  when two mobile agents x and y interact do
10:   execute Count_P
```

5 Conclusion and Perspectives

This paper studies a strong form of symmetry breaking, giving distinct names to indistinguishable agents in population protocols. It provides a comprehensive overview of the results in terms of possibility, impossibility and space optimality, and some insights on the trade-offs between criteria (global vs. weak, symmetric vs. asymmetric, need of a leader, initialization).

A continuation of this work could be the study of the time complexity aspects of naming and, overall, of the trade-offs between time and space. Another perspective would be to consider other forms of symmetry breaking (compact naming, leader election, coloring, two-hop coloring, majority, etc.), under constraints of optimal memory space and requirements of fault-tolerance.

References

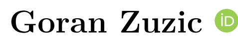
1. D. Angluin, J. Aspnes, Z. Diamadi, M. J. Fischer, and R. Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, 18(4):235–253, 2006. doi:10.1007/s00446-005-0138-3.
2. D. Angluin, J. Aspnes, D. Eisenstat, and E. Ruppert. The computational power of population protocols. *Distributed Computing*, 20(4):279–304, 2007. doi:10.1007/s00446-007-0040-2.
3. D. Angluin, J. Aspnes, M. J. Fischer, and H. Jiang. Self-stabilizing population protocols. *ACM Trans. Auton. Adapt. Syst.*, 3(4), 2008. doi:10.1145/1452001.1452003.
4. J. Beauquier, J. Burman, S. Clavière, and D. Sohier. Space-Optimal Counting in Population Protocols. In *DISC*, pages 631–646, 2015. doi:10.1007/978-3-662-48653-5_42.
5. J. Beauquier, J. Clement, S. Messika, L. Rosaz, and B. Rozoy. Self-Stabilizing Counting in Mobile Sensor Networks with a base station. In *DISC*, pages 63–76, 2007.
6. O. Bournez, J. Chalopin, J. Cohen, X. Koegler, and M. Rabie. Population Protocols that Correspond to Symmetric Games. *IJUC*, 9(1-2):5–36, 2013. URL: http://www.oldcitypublishing.com/journals/ijuc-home/ijuc-issue-contents/ijuc-volume-9-number-1-2-2013/ijuc-9-1-2-p-5-36/.
7. O. Bournez, J. Cohen, and M. Rabie. Homonym Population Protocols. *Theory Comput. Syst.*, 62(5):1318–1346, 2018. doi:10.1007/s00224-017-9833-2.
8. J. Burman, J. Beauquier, and D. Sohier. Space-Optimal Naming in Population Protocols. Research report, LRI, Université Paris Sud, 2019. URL: https://hal.inria.fr/hal-01790517.
9. S. Cai, T. Izumi, and K. Wada. How to Prove Impossibility Under Global Fairness: On Space Complexity of Self-Stabilizing Leader Election on a Population Protocol Model. *Theory Comput. Syst.*, 50(3):433–445, 2012. doi:10.1007/s00224-011-9313-z.
10. I. Chatzigiannakis, O. Michail, S. Nikolaou, A. Pavlogiannis, and P. G. Spirakis. Passively mobile communicating machines that use restricted space. *Theor. Comput. Sci.*, 412(46):6469–6483, 2011. doi:10.1016/j.tcs.2011.07.001.
11. C. Cooper, A. Lamani, G.i Viglietta, M. Yamashita, and Y. Yamauchi. Constructing self-stabilizing oscillators in population protocols. *Inf. Comput.*, 255:336–351, 2017. doi:10.1016/j.ic.2016.12.002.
12. C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, and E. Ruppert. When Birds Die: Making Population Protocols Fault-Tolerant. In *DCOSS*, pages 51–66, 2006. doi:10.1007/11776178_4.
13. E. W. Dijkstra. Self-stabilizing Systems in Spite of Distributed Control. *Commun. of the ACM*, 17(11):643–644, November 1974.
14. S. Dolev, A. Israeli, and S. Moran. Self-Stabilization of Dynamic Systems Assuming Only Read/Write Atomicity. *DC*, 7(1):3–16, 1993.
15. R. Guerraoui and E. Ruppert. Names Trump Malice: Tiny Mobile Agents Can Tolerate Byzantine Failures. In *ICALP (2)*, pages 484–495, 2009. doi:10.1007/978-3-642-02930-1_40.
16. T. Izumi, K. Kinpara, T. Izumi, and K. Wada. Space-efficient self-stabilizing counting population protocols on mobile sensor networks. *Theor. Comput. Sci.*, 552:99–108, 2014. doi:10.1016/j.tcs.2014.07.028.
17. H. Jiang. *Distributed Systems of Simple Interacting Agents*. PhD thesis, Yale University, 2007.
18. Y. Sudo, T. Masuzawa, A. K. Datta, and L. L. Larmore. The Same Speed Timer in Population Protocols. In *ICDCS*, pages 252–261, 2016. doi:10.1109/ICDCS.2016.82.
19. G. Tel. *Introduction to Distributed Algorithms (2nd ed.)*. Cambridge University Press, 2000.
20. H. Yasumi, F. Ooshita, K. Yamaguchi, and M. Inoue. Constant-Space Population Protocols for Uniform Bipartition. In *OPODIS*, pages 19:1–19:17, 2017. doi:10.4230/LIPIcs.OPODIS.2017.19.

Erasure Correction for Noisy Radio Networks

Keren Censor-Hillel
Technion, Haifa, Israel
ckeren@cs.technion.ac.il

Bernhard Haeupler
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA
haeupler@cs.cmu.edu

D. Ellis Hershkowitz
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA
dhershko@cs.cmu.edu



Goran Zuzic
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA
gzuzic@cs.cmu.edu

Abstract

The radio network model is a well-studied model of wireless, multi-hop networks. However, radio networks make the strong assumption that messages are delivered deterministically. The recently introduced noisy radio network model relaxes this assumption by dropping messages independently at random.

In this work we quantify the relative computational power of noisy radio networks and classic radio networks. In particular, given a non-adaptive protocol for a fixed radio network we show how to reliably simulate this protocol if noise is introduced with a multiplicative cost of $\text{poly}(\log \Delta, \log \log n)$ rounds where n is the number nodes in the network and Δ is the max degree. Moreover, we demonstrate that, even if the simulated protocol is not non-adaptive, it can be simulated with a multiplicative $O(\Delta \log^2 \Delta)$ cost in the number of rounds. Lastly, we argue that simulations with a multiplicative overhead of $o(\log \Delta)$ are unlikely to exist by proving that an $\Omega(\log \Delta)$ multiplicative round overhead is necessary under certain natural assumptions.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Theory of computation → Distributed algorithms; Theory of computation → Graph algorithms analysis

Keywords and phrases radio networks, erasure correction, noisy radio networks, protocol simulation, distributed computing models

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.10

Related Version A full version of the paper is available at `https://arxiv.org/abs/1805.04165`.

Funding Supported in part by NSF grants CCF-1527110, CCF-1618280, CCF-1814603, CCF-1910588, NSF CAREER award CCF-1750808 and a Sloan Research Fellowship.
Keren Censor-Hillel: Supported in part by the Israel Science Foundation (grant 1696/14), the Binational Science Foundation (grant 2015803) and the European Union's Horizon 2020 Research and Innovation Programme under grant agreement no. 755839.
Goran Zuzic: Supported in part by DFINITY Scholarship Program.

1 Introduction

The study of distributed graph algorithms provides precise mathematical models to understand how to accomplish distributed tasks with minimal communication. A classic example of such a model is the radio network model of [13]. Radio networks use synchronous rounds of communication and were designed to model the collisions that occur in multi-hop, wireless networks. However, radio networks make the strong assumption that, provided no collisions occur, messages are guaranteed to be delivered.

© Keren Censor-Hillel, Bernhard Haeupler, D. Ellis Hershkowitz, and Goran Zuzic;
licensed under Creative Commons License CC-BY

33rd International Symposium on Distributed Computing (DISC 2019).
Editor: Jukka Suomela; Article No. 10; pp. 10:1–10:17

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

This assumption is overly optimistic for real environments in which noise may impede communication, and so previous work of [11] introduced the noisy radio network model where messages are randomly dropped with a constant probability. This prior work demonstrated that the runtime of existing state-of-the-art broadcast protocols deteriorates significantly in the face of random noise. Furthermore, it showed how to design efficient broadcasting protocols that are robust to noise.

However, it has remained unclear how much more computationally powerful radio networks are than noisy radio networks. In particular, it was not known how efficiently an arbitrary protocol from a radio network can be *simulated* by a protocol in a noisy radio network. A simulation with little overhead would demonstrate that radio networks and noisy radio networks are of similar computational power. However, if simulation was necessarily expensive then radio networks would be capable of completing more communication-intensive tasks than their noisy counterparts.

A simple observation regarding the relative power of these two models is that any polynomial-length radio network protocol can be simulated at a multiplicative cost of $O(\log n)$ rounds where n is the number of nodes in the network. In particular, we can simulate any protocol from the non-noisy setting by repeating every round $O(\log n)$ times. A standard Chernoff and union bound argument show that every message sent by the original protocol is successfully sent with high probability. However, a multiplicative $O(\log n)$ is a significant price to pay for noise-robustness for many radio network protocols. For example, the optimal known-topology message broadcast protocol of [25] uses only $O(D + \log^2 n)$ rounds, while the topology-oblivious Decay protocol of [6] takes $O(D \log n + \log^2 n)$ where D is the diameter of the network. Moreover, a dependence on a global property of the network – the number of nodes in the graph – is excessive for correcting for a local issue – faults.

The simple solution from above *globally synchronizes* nodes by forcing all nodes to simulate the same round for $O(\log n)$ repetitions. A natural question is can one more efficiently simulate a noisy protocol by enforcing only *local synchronization*. In this paper we show how to use local synchronization to efficiently simulate radio network protocols in a noisy setting.

The local synchronization technique we use is as follows. Suppose each node tracked the round in the original protocol up to which it has successfully simulated; we call this round a node's *virtual round*. In our simulation in each round each node simulate the virtual round of its neighbor which has successfully simulated the fewest total rounds; i.e. each node simulates the virtual round of its neighbor with the largest "delay", thereby "helping" it. Such a simulation is local as – unlike the above simple simulation – nodes in distant parts of the network may simulate different rounds of the original protocol. Moreover, if a node has successfully simulated fewer rounds than all of its neighbors (and all of its neighbors' neighbors) then after one more round of simulation it will successfully simulate a new round of the original protocol if no random faults occur.

However, there are at least three notable challenges in implementing and proving the efficiency of such a local-synchronization-based simulation.

1. First, one must show that locally synchronizing nodes yields a fast simulation. A priori, it is not clear that locally synchronizing nodes provides an advantage over the simple global synchronization strategy.
2. Second, one must deal with the fact that local synchronization requires nodes to determine the minimal virtual rounds of its neighbors. In particular, nodes cannot easily compute the virtual rounds of their neighbors without communicating: When node v simulates a round of the original protocol and broadcasts should it assume all of its neighbors have

now successfully simulated this round? If a random fault occurred at a receiver then clearly v should not but v has no easy means of determining whether or not any such faults occurred. One must, therefore, determine how nodes can efficiently determine the minimal virtual round of their neighbors.

3. Lastly, one must overcome the fact that not receiving a message is indistinguishable from a randomly dropped message. In particular, a node can interpret not receiving a message in a simulated round as indicating either (1) that it receives no message in the simulated round of the original protocol or (2) that it does receive a message in this simulated round but a random fault dropped this message. Thus, it is not clear how nodes ought to increment their virtual rounds when they do not receive a message. On the one hand, failing to increment a node's virtual round in the former case could needlessly slow down the simulation. On the other hand, if a node incorrectly decides it does not receive a message in a round, its idea of what messages it received will diverge from that of its neighbor who tried to send it a message. This divergence, in turn, may compound into further errors later in the simulation.

1.1 Our Contributions

In this work we present solutions for these challenges which demonstrate that, by using local synchronization, radio network protocols can be simulated by noisy radio networks with a multiplicative dependence on Δ, the maximum degree of the network. Thus, we demonstrate that, for low-degree networks, noisy radio networks are essentially of the same computational power as radio networks. Moreover, we demonstrate that this dependence is more or less tight in two natural settings.

We give our simulations in three increasingly difficult settings, each of which introduces one of the above three challenges. In particular, our first model focuses on the first challenge, our second focuses on the first two challenges and our last model deals with all three challenges. We briefly mention our techniques here but defer more thorough intuition regarding our simulation techniques until Section 4, which is after we have more formally defined our problem.

Local Progress Detection. As a warmup we begin by providing a simulation with $O(\log \Delta)$ multiplicative round overhead in the setting where nodes have access to "*local progress detection*". Roughly, local progress detection enables nodes to know when they experience a random fault and also the virtual rounds of their neighbors. Notice that in this setting challenges 2 and 3 are non-issues: if each node has local progress detection, they can easily determine neighbors' delays and distinguish not receiving a message in the original protocol from a randomly dropped message. The key technique we introduce for the first challenge is a concentration-inequality-type result based on what we call "blaming chains". (Section 5)

Non-Adaptive Protocols. Next, we provide simulations for *non-adaptive* protocols. Roughly, non-adaptive protocols are protocols in which nodes know a priori the rounds in which they receive messages. In this setting, our simulations achieve a multiplicative $\text{poly}(\log \Delta, \log \log n) = O(\log^3 \Delta \cdot \log \log n \cdot \log \log \log n)$ round overhead *without local progress detection*. As we argue in Section 6, a number of well-known protocols (e.g. the optimal broadcast algorithm of [25]) are non-adaptive. Notice that in this setting we need not deal with challenge 3 above since nodes can always distinguish a dropped message from a round in which they receive no messages because they know the rounds of the original protocol in which they receive messages. In this setting, we leverage non-adaptiveness of protocols to overcome

challenge 2. In particular, we use a distributed binary search which carefully silences nodes that search over divergent ranges to inform nodes of the minimal virtual round of their neighbors. Moreover, we keep the range over which the binary search must search tractable by slowing down nodes that make progress too quickly. (Section 6)

General Protocols. Lastly, we show how to deal with all three challenges at once by giving simulations for arbitrary radio network protocols with a multiplicative $O(\Delta \log^2 \Delta)$ overhead. To overcome challenge 3 we have nodes exchange "tokens" with all neighbors in every round to preclude the possibility of a dropped message going unnoticed. (Section 7)

Lower Bounds. We also show that our simulations for non-adaptive protocols are likely optimal: We show that two natural classes of simulations necessarily use $\Omega(\log \Delta)$ multiplicatively many more rounds than the protocols which they simulate. In particular, a simulation that either (1) does not use network coding, or (2) sends information in the same way as the original protocol requires $\Omega(\log \Delta)$ multiplicatively many more rounds than the original protocol. We also give a construction which we believe gives an unconditional $\Omega(\log \Delta)$ simulation overhead. (Section 8)

2 Related Work

Several models have been studied to understand robust communication in models similar to the radio network model. [16] introduced the noisy broadcast model, which also assumes random errors, and focuses on studying the computation of functions of the inputs of nodes [18, 40, 26, 38]. The model of [16] differs from the noisy radio network model in that it assumes a complete communication network and single-bit transmissions. [42] introduced a similar model which again assumes single-bit transmissions and also does not have collisions as the noisy radio network model does. Several papers have been written on notions of noisy radio networks which assume the network admits geometric structure [33, 34]. There have also been several papers on noisy single-hop radio networks. See either [24] for a study of an adversarial model or [15] for a nice study of a random noise model. Another model which captures uncertainty is the *dual graph* model [35, 10, 36, 21, 23], in which an adversary chooses a set of unreliable edges in each round. Recent work [29] has also made use of several of our techniques for variants of the SINR model.

There has also been extensive work on two-party interactive communication in the presence of noise [44, 27, 19]. In this setting, Alice and Bob have some conversation in mind they would like to execute over a noisy channel; by adding redundancy they hope to hold a slightly longer conversation from which they can recover the original conversation outcome even when a fraction of the coded conversation is corrupted. Multi-party generalizations of this problem have also been studied [8]. The noisy radio network model can be seen as a radio network analogue of these interactive communication models in which erasures occur rather than corruptions.

Lastly, work on MAC layers has sought to provide abstractions for algorithms that hide low-level uncertainty in wireless communication [22, 35, 43, 30]. Radio networks differ from MAC layers as in radio networks it is not required that a sender receive an acknowledgment from a receiver.

Since its introduction by [13], the classic radio network model has attracted wide attention from researchers. The survey of [41] is an excellent overview of this research area. Here we focus the radio network literature that relates to our work. Much of previous work for the noisy

radio network model focused on broadcast [31]. For the classic model, [6] gave a single-message broadcast algorithm for a known topology, which completes in $O(D\log n+\log^2 n)$ rounds. [11] shows that this protocol is robust to noise, completing in $O(\frac{\log n}{1-p}(D+\log n+\log\frac{1}{\delta}))$ rounds, with a probability of failure of at most δ. In the classic model, single-message broadcast was then improved by [25] and [32], who showed that in the case of a known topology, $O(D+\log^2 n)$ rounds suffice. [11] shows that this protocol is *not* robust to noise, requiring in expectation $\Theta(\frac{p}{1-p}D\log n+\frac{1}{1-p}D)$ rounds, for broadcasting a message along a path of length D. They showed an alternative protocol, completing in $O(D+\log n\log\log n(\log n+\log\frac{1}{\delta}))$ rounds, with a probability of failure of at most δ. For an unknown topology in the classic model, [14] give a protocol completing in $O(D\log(n/D)+\log^2 n)$ rounds, which is optimal, due to the $\Omega(\log^2 n)$ and $\Omega(D\log(n/D))$ lower bounds of [2] and [37], respectively. [20] give a $O(D+\operatorname{poly}\log n)$-round protocol that uses *collision detection*.

Lastly, simulations of models of distributed computation by other models of distributed computation is a foundational aspect of distributing computing, dating back to the 80's–90's with many simulations of various shared memory primitives, faults, and more (see a wide variety in, e.g., [3]). It is also a focus for message-passing models with different features, being the motivation for synchronizers [4, 5], and additional simulations [12, 9].

3 Model and Assumptions

In this section we formally define the classic radio network model, the noisy radio network model and discuss various assumptions we make throughout the paper.

Radio Networks. A multi-hop radio network, as introduced in [13], consists of an undirected graph $G=(V,E)$ with $n:=|V|$ nodes. Communication occurs in synchronous rounds: in each round, each node either broadcasts a single message containing $\Theta(\log n)$ bits to its neighbors or listens. A node receives a message in a round if and only if it is listening and *exactly* one of its neighbors is transmitting a message. If two or more neighbors of node v transmit in a single round, their transmissions *collide* at v and v does not receive anything. We assume no collision detection, meaning a node cannot differentiate between when none of its neighbors transmit a message and when two or more of its neighbors transmit a message. Nodes are assumed to have unbounded computation, though all of the protocols in our paper use polynomial computation.

We use the following notational conventions throughout this paper when referring to radio networks. Let $\operatorname{dist}(v,w)$ be the hop-distance between u and v in G, let $\Gamma(v)=\{w\in V:\operatorname{dist}(v,w)\leq 1\}$ denote the 1-hop neighborhood of v, and let $\Gamma^{(k)}(v)=\{w\in V:\operatorname{dist}(v,w)\leq k\}$ denote the k-hop neighborhood of v.

Noisy Radio Networks. A multi-hop noisy radio network, as introduced in [11], is a radio network with random erasures. In particular, it is a radio network where node v receives a message in a round if and only if it is listening, exactly one of its neighbors is broadcasting and a *receiver fault* does not occur at v. Receiver faults occur at each node and in each round independently with constant probability $p\in(0,1)$. As p will be treated as a constant it will be suppressed in our O and Ω notation. We assume that in a given round a node cannot differentiate between a message being dropped because of a fault, i.e. a collision, and all of its neighbors remaining silent.

We consider this model because independent receiver faults model transient environmental interference such as the capture effect [39].[1] Moreover, we consider an *erasure* model – entire messages are dropped – rather than a corruption model – messages are corrupted at the bit level – because, in practice, wireless communication typically incorporates error correction and checksums that can guard against bit corruptions [17]. The noisy radio network model can also be seen as modeling those cases when this error correction fails and the message cannot be reconstructed and is therefore effectively dropped.

Protocols. A protocol governs the broadcast and listening behavior of nodes in a network. In particular, a protocol tells each node in each round to listen or what message to broadcast based on the node's history. This history includes the messages the node received, when it received them and its initial private input. We assume that this private input includes the number of nodes, n, the maximum degree, Δ, the receiver fault probability, p, a private random string, and any other data nodes have as input in a protocol. Formally, a **history** for node v is an $H \in \mathcal{H}$ where $\mathcal{H}$ is all valid histories. $\mathcal{H} = \{(i, R) : i \in \mathcal{I}, R \subseteq M \times \mathbb{N}\}$ where $\mathcal{I}$ is all valid private inputs, R gives what messages v has received in each round and $M = \{0,1\}^{O(\log n)}$ is all $O(\log n)$ bit messages a node could receive in a single round. Formally, a **protocol** P of length T is a function $P : V \times [T] \times \mathcal{H} \to \{\mathit{listen}\} \cup M$ where $P(v, t, H) = \mathit{listen}$ indicates that v listens in round t and $P(v, t, H) = m \in M$ indicates that v broadcasts message $m \in M$ in round t. We let $|P| := T$ stand for the length of a T round protocol. In this paper we assume that T is at most $\text{poly}(n)$.

Simulating a Protocol in the Noisy Setting. We say that protocol P' successfully **simulates** P if, after executing P' in the noisy setting, every node in the network can reconstruct the messages it would receive if P were run in the faultless setting. In particular, for any set of private input $I \in \mathcal{I}^{|V|}$ – letting $H(v, I)$ be v's history after running P with private inputs I – it must hold that after running P' with private inputs I, every v can compute $H(v, I)$. Note that nodes running P' can send messages not sent by P or send messages in a different order than they do in P. We call P the **original protocol** and P' the **simulation protocol**. We measure the efficacy of P' as the limiting ratio of $\frac{|P'|}{|P|}$ when T is sufficiently large and n goes to infinity, which we call the **multiplicative overhead** of a simulation.

4 Techniques Overview and Our Formal Results

As earlier mentioned, we show how to simulate a faultless protocol P in three noisy settings of increasing difficulty. Throughout our simulation results, we use the notion of a **virtual round** of node v, t_v, which tracks how many rounds of P node v has successfully simulated. We say that a node v is **most delayed** in a set of nodes U when $t_v \le \min_{w \in U} t_w$. All three simulations roughly work by having nodes first exchange their virtual rounds with their neighbors. Nodes then locally synchronize by simulating the virtual round of their most delayed neighbor. Our simulations differ in how each defines a virtual round and how nodes learn virtual rounds.

As a warmup and to study what sort of overhead local synchronization enables, we consider the setting where nodes have access to "local progress detection". Recall that local progress detection gives nodes oracle access to the virtual rounds of their neighbors as well as when faults occur. We show the following theorem which demonstrates that a $O(\log \Delta)$ overhead is possible in this setting.

[1] In contrast, a sender faults model in which an entire broadcast by a node is dropped might model hardware failures where independence of faults would be a poor modeling choice.

▶ **Theorem 1.** *Let P be a general protocol of length T for the faultless radio network model. P can be simulated in the noisy radio setting using local progress detection in $O(T \log \Delta + \log n + k)$ rounds with probability at least $1 - \exp(-k)$ for any $k \geq 0$.*

Blaming Chain Intuition. To prove the above theorem we use the idea of a blaming chain to argue that local synchronization enables small simulation overhead. Since every node simulates the round of its most delayed neighbor, a node v will have all neighbors simulate its virtual round when its virtual round is minimal among virtual rounds of nodes in $\Gamma^{(2)}(v)$. Moreover, as we will show, once v is most delayed in $\Gamma^{(2)}(v)$ it does not take *too many* additional rounds for nodes to successfully simulate v's virtual round. Thus, if v takes many rounds to successfully simulate virtual round x, it must be because there was a $u \in \Gamma^{(2)}(v)$ that required many rounds to simulate virtual round $x - 1$. In this way v can blame its delay on u. Node u, in turn, can blame the fact that it required many rounds to simulate virtual round $x - 1$ on the fact that one of its 2-hop neighbors, say w, required many rounds to simulate virtual round $x - 2$ and so on. Thus, if node v is very delayed we can explain this delay by some such blaming chain. There are exponentially many possible such blaming chains, but – by way of concentration inequalities we prove – we show that long blaming chains can be shown to have exponentially small probability and so a union bound over all long blaming chains will allow us to show that no long blaming chain occurs.

We next show how to efficiently spread virtual round information without using progress detection. In particular, we show a $\text{poly}(\log \Delta, \log \log n)$ overhead for non-adaptive protocols where nodes know a priori the rounds of the original protocol they are sent messages without collision.[2]

▶ **Theorem 2.** *Let P be a non-adaptive protocol of length T for the faultless radio network model. P can be simulated in $O((T + \log n) \log^3 \Delta \log \log n \log \log \log n)$ rounds in the noisy radio setting with high probability by* MainNonAdaptive*.*

Progress Throttling and Distributed Binary Search Intuition. In addition to blaming chains, the main technique we use to prove the above result is progress throttling and a novel algorithm for binary search in noisy radio networks. Since in this setting nodes can no longer learn their neighbors' virtual rounds using local progress detection, our goal is to provide nodes an alternative means of learning their neighbors virtual rounds; namely, we use progress throttling and distributed binary search. Since virtual rounds can be as small as 1 and as large as the length of the entire protocol which is as large as $\text{poly}(n)$, the range over which we must binary search might seem to be as large as $\text{poly}(n)$; a binary search over this range would require a prohibitive $O(\log n)$ iterations. For this reason, we slow down nodes that make progress too quickly by only increasing their virtual round at most once after $\log \Delta$ simulated rounds. We show that this throttling keeps all virtual rounds within an additive $O(\log n)$ range. This, in turn, allows our binary search to require only $O(\log \log n)$ iterations.

Moreover, actually implementing a distributed binary search in a radio network presents technical challenges of its own. The natural solution we use for node v to learn the smallest virtual round of a neighbor is for v to repeatedly ask its neighbors if any of them have a virtual round below the midpoint of the binary search range. v then updates its binary

[2] When we write that an event occurs **with high probability (w.h.p.)**, we mean that it occurs with probability $1 - \frac{1}{n^c}$, and that the constant $c = \Theta(1)$ can be made arbitrarily large by changing the constants in the routines.

search parameters in the usual way. This strategy enables an efficient binary search in radio networks because to update its binary search parameters, v need only hear from at most one neighbor. However, such a strategy suffers from the following problem of divergent ranges. Suppose node v has a neighbor u. u might have neighbors that are not neighbors of v which influence the range over which u searches. As such u might end up searching over a different range than v. These divergent ranges may render v's binary search nonsensical: v might query u to learn if its virtual round is below v's binary search midpoint and u could respond with whether or not it is below the midpoint of an entirely different search range, namely u's search range. We overcome this issue by carefully marking nodes "silent" if they might interfere with other nodes' binary search. Specifically, we show that, given a node v whose virtual round is minimal in $\Gamma^{(2)}(v)$, a careful silencing of possibly deviating nodes causes nodes in $\Gamma(v)$ to always update their binary search parameters in exactly the same manner as v. Also, we show that, while nodes in $\Gamma^{(2)}(v) \setminus \Gamma(v)$ might update their parameters differently, our silencing ensures that these nodes never interfere with the binary search performed by nodes in $\Gamma(v)$.

Lastly, we describe how to simulate *any* protocol with a $O(\Delta \log^2 \Delta)$ multiplicative overhead.

▶ **Theorem 3.** *Let P be a general protocol of length T for the faultless radio network model.* MainGeneral *simulates P in the noisy radio setting in $O((T \log \Delta + \log n)\Delta \log \Delta)$ rounds w.h.p.*

Token Exchange Intuition. In addition to blaming chains, the main technique we use to show the above theorem is a token exchange strategy. Recall that the main challenge in the general setting is that even if a node knows its neighbors are simulating its virtual round, the node cannot tell if the absence of a message indicates that it receives no message in this round in the original protocol or that a random fault occurred. As such if a node does not have a message to deliver to its neighbor, we force it to send its neighbor a token indicating that it has no message to send. This allows v to distinguish between a random fault and a round in which it simply receives no message.

In Section 8 we give a formal statement of how simulations in two natural settings require $\Omega(\log \Delta)$ overhead but we give some intuition here as to why this might be true here.

Lower Bound Construction Intuition. Consider a star with degree Δ where the center node wants to send T messages to its neighbors. A noiseless protocol requires T rounds but if the center only sends the original messages (and not e.g. an error correcting code) and random faults occur then the center must send each message about $\Omega(\log \Delta)$ times to guarantee all messages are delivered. Likewise consider a complete bipartite graph where each node on the left wants to deliver a message to the right. The existence of collisions in radio networks means that effectively only one node on the left can broadcast per round and, like the star, every node must broadcast about $\Omega(\log \Delta)$ times to deliver its message if there are random faults.

5 Warmup: Simulation with Local Progress Detection

We begin by giving our simulation with $O(\log \Delta)$ multiplicative overhead in the setting where nodes have access to **local progress detection**.

To rigorously define local progress detection, we introduce our notion of how much simulation progress nodes have made, the **virtual round**. We say that a node v successfully completed round t if it has succeeded in taking the action in P that it takes in t: v successfully

broadcasts its message m of round t if every node in $\Gamma(v)$ either has received m or had a collision in the original protocol P at round t; v successfully completes its listening action of round t if v receives the message it receives in round t of P or a collision occurs at v in round t of P. We formally define the virtual round and local progress detection.

▶ **Definition 4** (Virtual Round). *The virtual round of node $v \in V$ in a given round of simulation P' is the smallest $t_v \in \mathbb{Z}_{\geq 1}$ such that v has not successfully completed round t_v of P.*

▶ **Definition 5** (Local Progress Detection). *In the noisy radio network model with local progress detection every node v knows the virtual round of every node $w \in \Gamma^{(2)}(v)$ in every simulation round.*

We now sketch a proof of the main theorem of this section; the full proof is in the full version of the paper.

Proof Sketch of Theorem 1. We begin by describing our simulation, P'. Each node v repeatedly does the following in each round of P'. Let u be the node in $\Gamma(v)$ with minimal t_u. v takes the action that it takes in round t_u of P. That is, v tries to "help" u by simulating its virtual round.

Let $v \in V$ and note that the virtual round t_v never decreases. Hence for $x \in \mathbb{Z}_{\geq 1}$ we can define $D_{v,x}$ as the earliest round in P' when $t_v \geq x$. Notice that $D_{v,T}$ just is the number of rounds that v takes to simulate all rounds of the original protocol P. Thus, it will suffice for us to argue that $D_{v,T}$ is not too large for every v. We begin by finding a recurrence relation on $D_{v,x}$ saying that v will advance soon after its 2-hop neighborhood $\Gamma^{(2)}(v)$ has virtual round at least x. In particular we show $D_{v,x} \leq \left(\max_{w \in \Gamma^{(2)}(v)} D_{w,x-1}\right) + Y_{v,x}$ where $Y_{v,x}$ is the maximum of at most Δ many geometric random variables. Next, we show that this recurrence shows that $D_{v,x}$ is given by the longest "blaming chain" – formal definition deferred to the full paper. We then apply a Chernoff-style tail bound (proved in the full version) to show that a fixed blaming chain has small length with very high probability. By union bounding over all blaming chains we have that the length of the maximum blaming chain must be small and therefore $D_{v,T}$ is small for every v. ◀

6 Simulation for Non-Adaptive Protocols

We now give our simulation results for non-adaptive protocols with $\text{poly}(\log \Delta, \log \log n)$ overhead.

▶ **Definition 6** (Non-Adaptive Protocol). *A non-adaptive protocol P is a protocol in which every node can determine if it receives a message in each round of P regardless of the private inputs of the nodes. Formally, each node v is given a list, M_v, of the rounds in which it receives a message without collision in P.*

We let M_v.GETNEXTROUND return the smallest round in M_v such that for every $r \in M_v$ such that $r < M_v$.GETNEXTROUND, v has received the message it receives in round r of P. Since in a non-adaptive protocol nodes know in which rounds they receive messages in P, nodes in a simulation can locally compute M_v.GETNEXTROUND. Also notice that even though non-adaptive protocols assume nodes know a priori when they are supposed to receive messages, nodes do not know the contents of these messages before receiving them.

We note that a number of well-known protocols are non-adaptive. For instance, assuming nodes know the network topology and use public randomness, the optimal broadcast algorithm of [25] is non-adaptive. In particular, under these assumptions nodes can compute the

broadcast schedule of their neighbors for this protocol. The same is true of the Decay protocol of [6]. Since broadcast is the most studied problem in radio networks, the fact that the state-of-the-art broadcast algorithm is non-adaptive would seem to make studying non-adaptive protocols worthwhile. Moreover, though requiring that nodes know the network topology and use public randomness may seem restrictive, in the context of simulations this assumption is actually quite weak: we can dispense with both assumptions by simply running any primitive that informs nodes of the network topology or shares randomness before our simulation is run at a one-time additive round overhead. Any primitive to learn the network topology or share randomness from the classic radio network setting coupled with the simple $O(\log n)$ simulation as described in Section 1 suffices here. This overhead is negligible if the simulated protocol is sufficiently long or we run many simulations on our network. Lastly, we note that it is often the case that nodes can even *efficiently* compute the broadcast schedule of their neighbors – see [28] – and so this one-time cost is often quite small.

To prove Theorem 2 we build upon the idea of the preceding section of using virtual rounds to locally synchronize nodes. However, in the current setting it is difficult for nodes to confirm when their broadcasts have succeeded, and so we must relax our definition of virtual rounds. In particular, for the remainder of this section we let the **virtual round** of each node be the minimum between the largest $t_v \in \mathbb{Z}_{\geq 1}$ such that v receives a message without collision in t_v and v has successfully *received* all messages it receives up to round $t_v - 1$ in P in our simulation and a throttling variable L. That is, the virtual round of v is $\min(M_v.\text{GETNEXTROUND}, L)$. L will be a slowly increasing value and, in this way, will slow down the progress of nodes by keeping their virtual rounds small, thereby keeping all virtual rounds within an additive $O(\log n)$ range.

The main subroutine of our simulation is LearnDelays (Algorithm 1). LearnDelays performs a binary search to inform each node of the virtual round of its most delayed neighbor. Initially the search range of every node is between $L - O(\log n)$ and L. In each iteration of LearnDelays nodes with a virtual round less than the mean of their binary search range are "active". If there is an active node within 0 or 1 hop of node v then node v lowers the upper bound of its binary search. If there are no active nodes within 2 hops of v then it raises its binary search lower bound. Otherwise there is an active node 2 hops from v – which intuitively means that v is not most delayed in $\Gamma^{(2)}(v)$ – in which case node v remains silent for the rest of the binary search so as to not interfere with the binary search of its neighbors. LearnDelays uses subroutine DistToActive to inform nodes of their nearest active node. The properties of LearnDelays are given by the following lemma. Details of DistToActive and a formal proof of the following lemma are deferred to the full version.

▶ **Lemma 7.** *Assume that $L - O(\log n) \leq t_v \leq L$ for all $v \in V$ and let v be a most delayed node in its 2-hop neighborhood. With constant probability the output of* LearnDelays *for every $w \in \Gamma(v)$ is t_v, i.e., every w will try to help v advance. The runtime of* LearnDelays *is $O(\log^2 \Delta \log\log n \log\log\log n)$ rounds.*

We now present the MainNonAdaptive simulation routine that simulates P in the noisy setting (Algorithm 2). We let $P(v, t)$ return the action taken by node v in round t of P. The properties of MainNonAdaptive are given by the following lemma.

▶ **Lemma 8.** *Assume that $L - O(\log n) \leq t_v < L$ for all $v \in V$ and let v be a most delayed node in its 2-hop neighborhood. After an innermost iteration of* MainNonAdaptive*, t_v will increase by one with at least constant probability. Moreover, the running time of each innermost iteration is $O(\log^2 \Delta \log\log n \log\log\log n)$ rounds.*

Algorithm 1 LearnDelays for node v.

Require: L, t_v
  $lo \leftarrow L - O(\log n)$; $hi \leftarrow L$
  while $lo \neq hi$ **do** ▷ repeats $O(\log \log n)$ rounds
    v is marked as "active" iff $t_v \leq \lfloor \frac{lo+hi}{2} \rfloor$ and v is not marked "silent"
    dist $\leftarrow$ DistToActive
    if dist is "=2" **then** mark v as "silent" until the end of LearnDelays
    if dist is "=0" or "=1" **then**
      $hi \leftarrow \lfloor \frac{lo+hi}{2} \rfloor$
    else
      $lo \leftarrow \lfloor \frac{lo+hi}{2} \rfloor + 1$
  return lo

Proof. Fix a v that is most delayed in its 2-hop neighborhood such that $t_v < L$. By definition of a virtual round and the fact that $t_v < L$ we have that $P(v, t_v)$ is a listening action where in round t_v of P it holds that v is sent a message without collision. By Lemma 7 after LearnDelays terminates with constant probability each 1-hop neighbor w will have $m_w = t_v$. Thus, every node in the 1-hop neighborhood of v will simulate round t_v, i.e. the one neighbor that has a message for v will broadcast its message and the rest of v's neighbors will be silent. This message will be successfully received with probability p which is constant and so t_v will be incremented with constant probability. The running time follows easily by summing runtimes. ◀

Algorithm 2 MainNonAdaptive for node v.

  $t_v \leftarrow 1$
  for $L = 1, 2, \ldots, T + O(\log n)$ **do** ▷ "Outermost iteration"
    for repeat $O(\log \Delta)$ times **do** ▷ "Innermost iteration"
      $m_v \leftarrow$ LearnDelays
      do action $P(v, m_v)$
      if $m_v = M_v$.getNextRound and v received a message **then**
        $t_v \leftarrow \min(L, M_v.\text{getNextRound})$ ▷ Throttle v

We end this section with a proof sketch of Theorem 2; the full proof is in the full version of the paper.

Proof Sketch of Theorem 2. We divide our simulated schedule into length $O(\log n)$ chunks. Next we argue by induction that after each $O(\log n)$ outermost iterations of MainNonAdaptive every node is simulating the same chunk. This allows us to apply to apply Lemma 8 and a blaming chain argument as in Theorem 1 to argue that the current chunk is correctly simulated by all nodes. ◀

7 General Protocol Simulation

Here, we provide our results for arbitrary protocols. Our approach gives a $O(\Delta \log^2 \Delta)$ multiplicative overhead.

▶ **Theorem 3.** *Let P be a general protocol of length T for the faultless radio network model.* MainGeneral *simulates P in the noisy radio setting in $O((T \log \Delta + \log n)\Delta \log \Delta)$ rounds w.h.p.*

Again, we build on the notion of a virtual round for this setting. Our main challenge in this setting is that even if a node knows its neighbors are simulating its virtual round, the node cannot tell if the absence of a message indicates that it receives no message in this round in the original protocol or that a random fault occurred. As such, in order for node v to advance its virtual round after hearing no messages from a neighbor, v must confirm that every neighbor was silent in the simulated round. Let P be the original protocol for the faultless setting. Define the **token** for a node v in round r of P to be either the message that v is sending in round r of P or an arbitrary message indicating "v is not broadcasting" if v is silent in round r of P. Next, we (re-)define the **virtual round** of a node v to be the largest $t_v \in \mathbb{Z}_{\geq 1}$ such that v successfully received *all tokens from all neighbors* for rounds $1, 2, \ldots, t_v - 1$ (for a total of up to $(t_v - 1)\Delta$ tokens).

Our simulation algorithm works in two phases: every node first informs its neighbors of its virtual round; next, nodes help the neighbor with the smallest t_v they saw by sharing the token for that round. We now present pseudocode for the ShareKnowledge routine which shares messages from a node with all of its neighbors and MainGeneral which simulates P in the noisy setting.

Algorithm 3 ShareKnowledge for node v.

Require: a message, B_v, that v wants to share

for $O(\Delta \log \Delta)$ rounds **do**

  v broadcasts B_v with probability $\frac{1}{\Delta}$, independently from other nodes

---

▶ **Lemma 9.** *After* ShareKnowledge *terminates, a fixed node v successfully receives messages from all its neighbors with probability at least $3/4$ and successfully sends its message to all neighbors with probability at least $3/4$. The running time of* ShareKnowledge *is $O(\Delta \log \Delta)$ rounds.*

Proof. Fix an arbitrary node v. Consider the event where v receives a message from a fixed neighbor w in a fixed iteration of ShareKnowledge. This event occurs iff w broadcasts and all other neighbors of v, namely $\Gamma(v) \setminus \{w, v\}$, do not broadcast. This occurs with probability that is at least $\frac{1}{\Delta}(1 - \frac{1}{\Delta})^{|\Gamma(v)\setminus\{w,v\}|} \geq \frac{1}{\Delta}(1 - \frac{1}{\Delta})^{\Delta} \geq \Omega(\frac{1}{\Delta})$ by $(1 - \frac{1}{x})^x = \Omega(1)$. The probability that v does not hear from w after $O(\Delta \log \Delta)$ iterations is $(1 - \Omega(\frac{1}{\Delta}))^{\Delta \log \Delta} \leq \exp(-\Omega(\log \Delta)) \leq \frac{1}{4\Delta}$. Union bounding over all $|\Gamma(v)| \leq \Delta$ possibilities for w we get that the probability of v not sharing knowledge with all neighbors is at most $1/4$. ◀

Algorithm 4 MainGeneral for node v.

$t_v \leftarrow 1$

for $O(T \log \Delta + \log n)$ **do**

  v runs ShareKnowledge(t_v)

  $m_v \leftarrow$ smallest value v receives from all nodes running ShareKnowledge

  ShareKnowledge(token for virtual round m_v)

  update t_v if v received all tokens for round t_v

▶ **Lemma 10.** *Let v be a most delayed node in its 2-hop neighborhood. After one* MainGeneral *loop iteration, t_v will increase by one with at least constant probability.*

Proofs of Lemma 10 and Theorem 3 are deferred to the full version of the paper.

8 Lower bounds

In this section we argue that an $\Omega(\text{poly} \log \Delta)$ multiplicative overhead in simulation is necessary in two natural settings. In the first setting the simulation is not permitted to use network coding [1]. In the second setting the simulation must respect information flow in the sense that we show a lower bound when the graph is directed. It follows that any simulation with constant overhead either uses network coding or does not respect information flow. We also give a construction which we believe could be used to show an $\Omega(\text{poly} \log \Delta)$ multiplicative overhead in simulation, even without any assumptions.

Additionally, we strengthen our lower bounds by proving them in the setting in which the simulation is granted "global control". Informally, global control eliminates the need for control messages by providing a centralized scheduler that can synchronize nodes based on how faults occur. The scheduler, however, cannot read the actual contents of the messages.

▶ **Definition 11** (Global Control). *We say that a noisy radio network has global control when (1) nodes know the network topology, (2) nodes learn which nodes broadcast in each round and at which nodes receiver faults occur in each round, and (3) all nodes have access to public randomness.*

It is not difficult to see that access to global control is sufficient to achieve the local progress detection of Section 5. Moreover, notice that our simulation from Section 5 with $O(\log \Delta)$ multiplicative overhead is both non-coding and respects information flow. As such, it is optimal for both settings.

8.1 Non-Coding Simulations

We now define a non-coding protocol and prove that simulations that do not use network coding suffer a $\Omega(\log \Delta)$ multiplicative overhead.

▶ **Definition 12** (Non-Coding). *We say that a simulation P' in the noisy setting, which is simulating a protocol P in the faultless setting, is non-coding if any message sent in P' is also sent in P (though possibly by different nodes).*

We consider an isolated star with degree Δ where the center node wants to send T messages to its neighbors. One can achieve a constant multiplicative overhead on this protocol by using an error correction code like Reed-Solomon [45]. However, the following lemma shows that if coding is not used no such overhead is possible.

▶ **Lemma 13.** *For any $T \geq 1$ and sufficiently large Δ there exists a faultless protocol of length T on the star network with $\Delta + 1$ nodes such that any non-coding simulation of the protocol in the noisy setting with constant success probability requires $\Omega(T \log \Delta)$ many rounds even if the simulation has access to global control.*

Proof. As noted, the network is a star with Δ leafs. Let r be the central node of the star. In our faultless protocol, r receives T private inputs $M_1, M_2, \ldots, M_T$ each of $\Theta(\log n)$ bits. r takes T rounds to broadcast each input, broadcasting M_i at round i.

Now consider a simulation of our protocol and assume for the sake of contradiction that it succeeds with constant probability. Let $C = C(p)$ be a constant such that $1 - p \geq \exp(-C)$ where $p = \Omega(1)$ is the fault probability of our noisy network. By the non-coding assumption, all messages sent by P' must be in the set $\{M_1, \ldots, M_T\}$. Denote by t_i the number of times r broadcasts M_i. For the sake of contradiction, assume that $\sum_{i=1}^{T} t_i \leq \frac{1}{100C} T \log \Delta$. Hence $\min_{i \in [T]} t_i \leq \frac{1}{100C} \log \Delta$ by an averaging argument. Let $i^* = \arg\min_{i \in [T]} t_i$. Notice that

the probability that a fixed node receives a message is independent of that of any other node since WLOG only r ever broadcasts. Therefore, the probability that a fixed node does not receive M_{i^*} is $(1-p)^{t_{i^*}} \geq (1-p)^{\frac{1}{100C}\log\Delta} \geq \exp(-C\frac{1}{100C}\log\Delta) \geq \Delta^{-1/100}$ by definition of C. Consequently, the probability that some node does not receive i^* is at least $1-(1-\Delta^{-1/100})^{\Delta} \geq 1-\exp(\Delta^{99/100})$ which tends to 1 as $\Delta \to \infty$. This contradicts our assumption that our simulation succeeds with constant probability. Therefore, no simulation protocol of length $\Omega(T\log\Delta)$ can deliver all messages with constant probability. ◀

8.2 Simulations that Respect Information Flow

In this section we show how any simulation with less than a $\Omega(\text{poly}(\log\Delta))$ multiplicative overhead must route information along different paths than those in the faultless setting. In particular, we show that a $\Omega(\text{poly}(\log\Delta))$ lower bound holds in a directed network where information in any simulation flows just as it does in the noiseless setting.

▶ **Lemma 14.** *Let $G=(L\cup R,E)$ be a complete directed bipartite graph with $|L|=|R|=\Delta$ that has an arc (l,r) for all $l\in L, r\in R$. There exists a protocol P of length Δ for the faultless setting on the directed network G such that any protocol that works in the noisy setting with constant success probability requires $\Omega(\Delta\log\Delta)$ rounds. This bound holds even in the global control setting.*

Proof. Let $L=\{l_1,l_2,\ldots,l_\Delta\}$ and $R=\{r_1,r_2,\ldots,r_\Delta\}$ be the set of nodes on both sides of the partition. Every node $l_i\in L$ gets private input M_i and needs to broadcast it to all nodes in R. In the faultless protocol P, l_i broadcasts M_i in round $i\in[\Delta]$.

Now consider a noisy protocol P'. Since G is directed, the only node from L that has knowledge of M_i is l_i. Moreover, we can assume without loss of generality that in any one round of P' at most one node in L broadcasts, since if this were not the case either no node in R would be sent a message or a collision would occur at every node in R. Let t_i be the number of rounds in P' that l_i broadcasts M_i. For the sake of contradiction, assume that the length of P' is $c\Delta\log\Delta$ for a sufficiently small constant $c>0$. Therefore, $\sum_{i=1}^{\Delta} t_i \leq c\Delta\log\Delta$ and there exists $i^*\in[\Delta]$ such that $t_{i^*}\leq c\log\Delta$. Since M_{i^*} can only be broadcasted from l_{i^*}, M_{i^*} is broadcasted at most $c\log\Delta$ times by an averaging argument. Thus, we have a star with central node l_{i^*} and leaves given by R with the assumption that $c\log\Delta$ rounds suffices to spread a message from l_{i^*} to every node in R. The remainder of the proof is identical to the strategy given in Lemma 13 and hence is omitted. ◀

8.3 Unconditional Lower Bound Hypothesis

We do not believe there exists an $o(\text{poly}(\log\Delta))$ multiplicative overhead simulation, even for non-adaptive protocols, and in this section we put forward a candidate hard example that might be used to prove this claim.

Construction. Let G be a bipartite network with partition (L,R), where $|L|=|R|=n$. Divide the nodes in L into Δ groups of size $\frac{n}{\Delta}$, namely $L^1,\ldots,L^\Delta$. Let $l_1^i,\ldots,l_{n/\Delta}^i$ be the nodes in L^i. We repeat the following for $t=1,2,\ldots,\Delta$ iterations: pick a fresh independent permutation $\pi:[n]\to[R]$ and divide R into Δ groups of size n/Δ according to π. Specifically, let $R^1=\{\pi(1),\pi(2),\ldots,\pi(n/\Delta)\}, R^2=\{\pi(n/\Delta+1),\ldots,\pi(2n/\Delta)\}$, ..., $R^\Delta=\{\pi(n-n/\Delta+1),\ldots,\pi(n)\}$. Note that the grouping R changes between iterations unlike L which remains fixed. Fully connect l_t^i to all the nodes in R^i for all $i\in[\Delta]$.

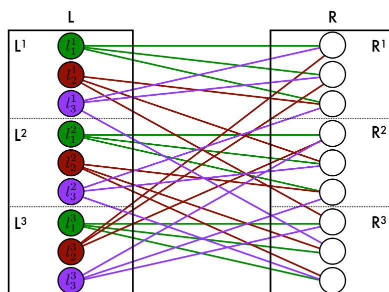


Figure 1 A particular sample of our construction which we believe could help prove an unconditional lower bound. R_i labeled according to the first iteration of our construction. $\Delta = 3$, $n = 9$. Nodes (and incident edges) colored according to the round in which they broadcast in the faultless protocol.

Why we believe this protocol is hard. Directly forwarding messages from a node $l \in L$ to each one of l's neighbors requires a multiplicative $\Omega(\log \Delta)$ overhead before all of the neighbors receive the message. Thus, if we assume there is a $o(\log \Delta)$ overhead protocol, there must be a large fraction of messages that are delivered indirectly. That is, many nodes in R receive many of the messages they need to simulate the original protocol from a different nodes than they do in the faultless protocol. However, indirectly delivering messages seems to require strictly more rounds than directly sending messages. Each $r \in R$ roughly wants to receive private input from a random subset of L. Therefore, if $r \in R$ receives a message indirectly from a neighbor $l \in L$, it is unlikely that the neighbors of l apart from r need this message to simulate the original protocol. Thus, while l delivers a message to r, it blocks all other neighbors of l from receiving messages they need to simulate the protocol. Lastly, we note that this problem roughly corresponds to a random instance of *index coding* [7], for which the bounds are currently not fully understood.

9 Future Work

Recall that the third challenge for local-synchronization-based simulations described in Section 1 is that nodes cannot distinguish between not receiving a message in the original protocol and a message being dropped by a random fault. Our general protocol solves this issue but only through a costly subroutine in which every node shares information with all of its neighbors, thereby incurring an $O(\Delta)$ overhead. We leave as an open question whether this challenge can be overcome with $\text{poly}(\log \Delta)$ overhead. "Backtracking" on faulty progress has been the subject of some interactive coding literature – see [27] – and will likely prove insightful on this front.

As a final direction for future work we note that many of our techniques apply to simulations of faulty versions of other models of distributed computing. For instance, applying the techniques of our general protocol simulation to a receiver fault version of CONGEST almost immediately yields a simulation of CONGEST with receiver faults by CONGEST with a $O(\log \Delta)$ multiplicative round overhead. We expect further applications.

References

1. R. Ahlswede, Ning Cai, S. Y.R. Li, and R. W. Yeung. Network Information Flow. *IEEE Transactions on Information Theory*, 46(4):1204–1216, September 2006.
2. Noga Alon, Amotz Bar-Noy, Nathan Linial, and David Peleg. A lower bound for radio broadcast. *Journal of Computer and System Sciences*, 43(2):290–298, 1991.
3. Hagit Attiya and Jennifer Welch. *Distributed Computing. Fundamentals, Simulations, and Advanced Topics.* McGraw-Hill, 1998.
4. Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM (JACM)*, 32(4):804–823, October 1985.
5. Baruch Awerbuch and David Peleg. Network synchronization with polylogarithmic overhead. In *Annual ACM Symposium on Theory of Computing (STOC)*, 1990.
6. Reuven Bar-Yehuda, Oded Goldreich, and Alon Itai. On the time-complexity of broadcast in multi-hop radio networks: An exponential gap between determinism and randomization. *Journal of Computer and System Sciences*, 45(1):104–126, 1992.
7. Ziv Bar-Yossef, Yitzhak Birk, TS Jayram, and Tomer Kol. Index coding with side information. *IEEE Transactions on Information Theory*, 57(3):1479–1494, 2011.
8. Mark Braverman, Klim Efremenko, Ran Gelles, and Bernhard Haeupler. Constant-rate coding for multiparty interactive communication is impossible. *Journal of the ACM (JACM)*, 65(1):4, 2017.
9. Keren Censor-Hillel, Ran Gelles, and Bernhard Haeupler. Making Asynchronous Distributed Computations Robust to Channel Noise. In *Innovations in Theoretical Computer Science Conference (ITCS)*, pages 50:1–50:20, 2018.
10. Keren Censor-Hillel, Seth Gilbert, Fabian Kuhn, Nancy A. Lynch, and Calvin C. Newport. Structuring unreliable radio networks. *Distributed Computing*, 27(1):1–19, 2014.
11. Keren Censor-Hillel, Bernhard Haeupler, D Ellis Hershkowitz, and Goran Zuzic. Broadcasting in Noisy Radio Networks. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2017.
12. Keren Censor-Hillel, Bernhard Haeupler, Jonathan A. Kelner, and Petar Maymounkov. Rumor Spreading with No Dependence on Conductance. *SIAM Journal on Computing (SICOMP)*, 46(1):58–79, 2017.
13. I. Chlamtac and S. Kutten. On Broadcasting in Radio Networks - Problem Analysis and Protocol Design. *IEEE Transactions on Communications*, 33(12):1240–1246, December 1985.
14. Artur Czumaj and Wojciech Rytter. Broadcasting algorithms in radio networks with unknown topology. *Journal of Algorithms*, 60(2):115–143, 2006.
15. Klim Efremenko, Gillat Kol, and Raghuvansh Saxena. Interactive coding over the noisy broadcast channel. In *Annual ACM Symposium on Theory of Computing (STOC)*, pages 507–520. ACM, 2018.
16. Abbas El-Gamal. Open problems presented at the 1984 workshop on specific problems in communication and computation. *Sponsored by bell communication research*, 1984.
17. Moncef Elaoud and Parameswaran Ramanathan. Adaptive use of error-correcting codes for real-time communication in wireless networks. In *INFOCOM*, volume 2, pages 548–555, 1998.
18. Robert G. Gallager. Finding parity in a simple broadcast network. *IEEE Transactions on Information Theory*, 34(2):176–180, 1988.
19. Ran Gelles et al. Coding for interactive communication: A survey. *Foundations and Trends in Theoretical Computer Science*, 13(1–2):1–157, 2017.
20. Mohsen Ghaffari, Bernhard Haeupler, and Majid Khabbazian. Randomized broadcast in radio networks with collision detection. *Distributed Computing*, 28(6):407–422, 2015.
21. Mohsen Ghaffari, Bernhard Haeupler, Nancy A. Lynch, and Calvin C. Newport. Bounds on Contention Management in Radio Networks. In *Distributed Computing (DISC)*, pages 223–237, 2012.

22 Mohsen Ghaffari, Erez Kantor, Nancy Lynch, and Calvin Newport. Multi-message broadcast with abstract mac layers and unreliable links. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 56–65, 2014.

23 Mohsen Ghaffari, Nancy A. Lynch, and Calvin C. Newport. The cost of radio network broadcast for different models of unreliable links. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 345–354, 2013.

24 Seth Gilbert, Rachid Guerraoui, Dariusz R Kowalski, and Calvin Newport. Interference-resilient information exchange. In *INFOCOM*, 2009.

25 Leszek Gąsieniec, David Peleg, and Qin Xin. Faster communication in known topology radio networks. *Distributed Computing*, 19(4):289–300, 2007.

26 Navin Goyal, Guy Kindler, and Michael E. Saks. Lower Bounds for the Noisy Broadcast Problem. *SIAM Journal on Computing (SICOMP)*, 37(6):1806–1841, 2008.

27 Bernhard Haeupler. Interactive channel capacity revisited. In *Symposium on Foundations of Computer Science (FOCS)*, pages 226–235, 2014.

28 Bernhard Haeupler and David Wajc. A faster distributed radio broadcast primitive. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 361–370, 2016.

29 Magnus M Halldorsson and Tigran Tonoyan. Plain SINR is Enough! In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 127–136. ACM, 2019.

30 Majid Khabbazian, Dariusz R Kowalski, Fabian Kuhn, and Nancy Lynch. Decomposing broadcast algorithms using abstract MAC layers. *Ad Hoc Networks*, 12:219–242, 2014.

31 Dariusz R Kowalski and Andrzej Pelc. Time complexity of radio broadcasting: adaptiveness vs. obliviousness and randomization vs. determinism. *Theoretical Computer Science*, 333(3):355–371, 2005.

32 Dariusz R. Kowalski and Andrzej Pelc. Optimal Deterministic Broadcasting in Known Topology Radio Networks. *Distributed Computing*, 19(3):185–195, 2007.

33 Evangelos Kranakis, Danny Krizanc, and Andrzej Pelc. Fault-tolerant broadcasting in radio networks. In *Annual European Symposium on Algorithms (ESA)*, pages 283–294, 1998.

34 Evangelos Kranakis, Michel Paquette, and Andrzej Pelc. Communication in random geometric radio networks with positively correlated random faults. In *International Conference on Ad-Hoc Networks and Wireless*, pages 108–121. Springer, 2008.

35 Fabian Kuhn, Nancy Lynch, and Calvin Newport. The abstract MAC layer. In *Distributed Computing (DISC)*, pages 48–62, 2009.

36 Fabian Kuhn, Nancy A. Lynch, Calvin C. Newport, Rotem Oshman, and Andréa W. Richa. Broadcasting in unreliable radio networks. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 336–345, 2010.

37 Eyal Kushilevitz and Yishay Mansour. An Omega(D Log(N/D)) Lower Bound for Broadcast in Radio Networks. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 1993.

38 Eyal Kushilevitz and Yishay Mansour. Computation in Noisy Radio Networks. In *Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, volume 98, pages 236–243, 1998.

39 Krijn Leentvaar and Jan Flint. The capture effect in FM receivers. *IEEE Transactions on Communications*, 24(5):531–539, 1976.

40 Ilan Newman. Computing in Fault Tolerance Broadcast Networks. In *IEEE Conference on Computational Complexity (CCC)*, pages 113–122, 2004.

41 David Peleg. *Time-Efficient Broadcasting in Radio Networks: A Review*, pages 1–18. Springer Berlin Heidelberg, 2007.

42 Sridhar Rajagopalan and Leonard Schulman. A coding theorem for distributed computation. In *Annual ACM Symposium on Theory of Computing (STOC)*, pages 790–799, 1994.

43 Andréa W Richa and Christian cheideler. Jamming-Resistant MAC Protocols for Wireless Networks. *Encyclopedia of Algorithms*, pages 1–5, 2008.

44 Leonard J Schulman. Coding for interactive communication. *IEEE Transactions on Information Theory*, 42(6):1745–1756, 1996.

45 Stephen B. Wicker. *Reed-Solomon Codes and Their Applications*. IEEE Press, Piscataway, NJ, USA, 1994.

Reachability and Shortest Paths in the Broadcast CONGEST Model

Shiri Chechik
Tel Aviv University, Israel
shiri.chechik@gmail.com

Doron Mukhtar
Tel Aviv University, Israel
doron.muk@gmail.com

Abstract

In this paper we study the time complexity of the single-source reachability problem and the single-source shortest path problem for directed unweighted graphs in the Broadcast CONGEST model. We focus on the case where the diameter D of the underlying network is constant.

We show that for the case where $D = 1$ there is, quite surprisingly, a very simple algorithm that solves the reachability problem in 1(!) round. In contrast, for networks with $D = 2$, we show that any distributed algorithm (possibly randomized) for this problem requires $\Omega(\sqrt{n/\log n})$ rounds. Our results therefore completely resolve (up to a small polylog factor) the complexity of the single-source reachability problem for a wide range of diameters.

Furthermore, we show that when $D = 1$, it is even possible to get an almost $3-$approximation for the all-pairs shortest path problem (for directed unweighted graphs) in just 2 rounds. We also prove a stronger lower bound of $\Omega(\sqrt{n})$ for the single-source shortest path problem for unweighted directed graphs that holds even when the diameter of the underlying network is 2. As far as we know this is the first lower bound that achieves $\Omega(\sqrt{n})$ for this problem.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Distributed algorithms, Broadcast CONGEST, distance estimation, small diameter

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.11

1 Introduction

Reachability and shortest path are two of the most fundamental problems in graph algorithms. In this paper, we study the single-source reachability (SSR) problem and the single-source shortest path (SSSP) problem in the Broadcast CONGEST model of distributed computing.

The CONGEST model [17] is one of the most studied message-passing models in the field of distributed computing. In this model, a synchronized n-vertex communication network is modeled by an undirected graph N whose vertices correspond to the processors in this network and whose edges correspond to the communication links between them. Each vertex has a unique $O(\log n)$-bit identifier initially known only to itself and its neighbors in N. The vertices communicate in discrete rounds, where in each round each vertex receives the messages that were previously sent to it, performs some unbounded local computation and then sends messages of $O(\log n)$ bits to all or some of its neighbors. The vertices work together on some common task (such as computing distances in the network) and the complexity is measured by the number of communication rounds needed to complete this task. The Broadcast CONGEST model is a more restrictive variant of the CONGEST model where every vertex has to send (broadcast) the same message to all of its neighbors in each round.

In this paper we focus on directed and unweighted graphs. In the SSR problem, we are asked to identify all the vertices in a given graph G for which there is a directed path from some designated vertex s called the source. In the SSSP problem, we are further asked to

© Shiri Chechik and Doron Mukhtar;
licensed under Creative Commons License CC-BY

compute for each such vertex its distance (the number of edges in a shortest path) from the source s. In the CONGEST model as well as in other similar message-passing models, we assume that the communication network N is identical to the underlying graph of G (where G is the input graph for the SSR\SSSP problem). We also assume that the communication between the vertices is bi-directional (regardless of directions of the edges in G). Initially, each vertex in the network knows whether it is the source or not, and it also knows its set of incoming and outgoing edges in G. In the distributed SSR problem, each vertex has to determine whether it is reachable from the source or not, and in the distributed SSSP problem, each vertex has to determine its distance from the source.

Related Work

Distance computation problems (such as the SSSP problem) have been widely studied in many models of distributed computing. It is not hard to see that in many synchronous message-passing models, problems such as SSR and SSSP require $\Omega(D)$ rounds (where D is the diameter of the underlying network). While this lower bound can be easily matched when messages of unbounded size are allowed, the situation for models that require the messages to be of bounded size is far more involved.

In the CONGEST model, it is possible to solve the directed single-source reachability problem in $\tilde{O}(\sqrt{n}D^{1/4} + D)$ rounds with high probability [10]. Many variants of the SSSP problem (directed\undirected, exact\approximate etc.) were studied over the years (see, e.g., [16, 11, 2, 7, 9, 8]). In particular, for directed and weighted graphs, there is a randomized algorithm that solves the SSSP problem in $\tilde{O}(\sqrt{nD})$ rounds [8]. We note that many of the above mentioned algorithms (such as [10, 8]) actually work in the more restrictive Broadcast CONGEST model. Regarding lower bounds, Das Sarma et al. [6] showed that in the CONGEST model the time complexity of any (possibly randomized) algorithm for the directed single-source reachability problem is $\Omega(\sqrt{n/\log n})$. However, this lower bound was shown only for graphs of underlying diameter $\Omega(n^{\delta})$ for some $0 < \delta < 1/2$. For smaller diameters, similar but weaker lower bounds were shown e.g. $\Omega(\sqrt{n}/\log n)$ for graphs of underlying diameter $\Theta(\log n)$. The smallest constant diameter for which a non-trivial lower bound is known is 3 where it was shown to require $\Omega((n/\log n)^{1/4})$ rounds.

For the related all-pairs shortest path (APSP) problem, many algorithms with near-optimal complexities for the approximate version of this problem and for the case of unweighted graphs were developed over the years (e.g., [12, 14, 15, 16]). Recently, many algorithms with improved complexities for the case of weighted graphs were devised [7, 13, 1] culminating with the $\tilde{O}(n)$-time randomized algorithm of [3].

Our Results

In this paper we study the time complexity of the SSR problem and the SSSP problem (for directed unweighted graphs) in the Broadcast CONGEST model for networks of constant diameter. Specifically, we show that even for networks of diameter 2, any distributed algorithm (possibly randomized) for the SSR problem requires $\Omega(\sqrt{n/\log n})$ rounds. In contrast, we show that quite surprisingly for networks of diameter 1, this problem (or even the more general all-pairs reachability problem) can be solved deterministically in 1 round. Moreover, we show that for networks of diameter 1 one can compute in 2 rounds a $(3, 2)$-approximation for the APSP problem (for directed unweighted graphs), where by (α, β)-approximation we mean α multiplicative approximation and β additive approximation.

The algorithm for the approximate APSP problem (resp. for the all-pairs reachability problem) allows each vertex to compute a $(3,2)$-approximation for the distance between every pair of vertices in the graph (resp. determine reachability for every such pair). We note that if one can compute a $(2-\epsilon)$-approximation for the APSP problem (for some $1 \geq \epsilon > 0$) such that there is some vertex v that knows the computed estimation for every pair of vertices, then this vertex can recover the whole graph. This means that v must receive in this case $\Theta(n^2)$ bits of information from its neighbors (simply because there are $\Theta(2^{n^2})$ possible graphs on these vertices), but in each round, v can get at most $O(n \log n)$ bits from its neighbors and so $\Omega(n/\log n)$ rounds are required for solving this problem.

Our results show a large gap between networks of diameter 1 and 2. As upper bounds of $\tilde{O}(\sqrt{n})$ are already known for the SSR problem when the underlying network has constant or poly-logarithmic diameter (e.g., [10, 8]), we completely resolve (up to poly-log factors) the SSR problem when the diameter of the underlying network is constant or even poly-logarithmic. Our algorithms are very simple (we see this as a plus and not a minus). In addition, we show a stronger lower bound of $\Omega(\sqrt{n})$ for the SSSP problem for unweighted directed graphs in the Broadcast CONGEST model that holds even when the diameter of the underlying network is 2. As far as we know this is the first lower bound that achieves $\Omega(\sqrt{n})$ for this problem.

Further Related Work

A closely related model to the CONGEST when the underlying communication network has diameter 1 is the Congested Clique model. The Congested Clique model is a synchronous message-passing model in which the underlying communication network is the complete graph on n vertices but the graph G on which the solution needs to be obtained can be an arbitrary graph on n vertices (that is, each vertex initially knows its neighbors in G and can exchange messages of size $O(\log n)$ with any vertex in the graph even if they are not adjacent in G).

Censor-Hillel et al. [5] adapted parallel matrix multiplication algorithms to this model. Using these algorithms, they obtained better algorithms for subgraph detection and distance computation. In particular, they showed a $\tilde{O}(n^{1/3})$-round algorithm for solving the APSP problem for weighted directed graphs, and even more efficient algorithms for unweighted undirected graphs or distance approximation. Recently, it was shown [4] that the SSSP problem for weighted undirected graphs can be solved in $\tilde{O}(n^{1/6})$ rounds.

We note that for problems such as SSSP or APSP (for weighted graphs) the Congested Clique model is actually a special case of the CONGEST model when the diameter of the underlying network is 1. To see this, note that one can always transform the input graph G into a complete graph by adding edges of very large weight. Therefore, either one can show a constant upper bound for the weighted SSSP problem in the Congested Clique model which will be quite a breakthrough or our upper bound shows a separation between the SSR problem and the SSSP problem for directed weighted graphs of underlying diameter 1 (and even between the all-pairs reachability problem and the SSSP problem).

2 Preliminaries

In the following, we assume that all directed graphs are simple (i.e., they do not contain self-loops or multiple edges, but they may contain anti-parallel edges). For a graph H, we respectively denote by $V(H)$ and $E(H)$ its vertex set and edge set. The out-degree and in-degree of a vertex v in a directed graph H are denoted by $d_{out}(v)$ and $d_{in}(v)$, respectively.

For a directed graph H and a vertex v in H, we denote by $N_{out}(v)$ its set of outgoing neighbors, and by $N_{in}(v)$ its set of ingoing neighbors. Given a directed graph $H = (V, E)$ and a set $A \subseteq V$, we denote by A^c the set $V \setminus A$. The underlying diameter of a directed graph H is defined to be the diameter of its underlying graph. For a graph H and two vertices v and u in $V(H)$, we denote by $d(v, u, H)$ the distance from v to u in H. All logarithms in this paper are of base 2.

The rest of paper is organized as follows. In section 3, we show an algorithm that solves the all-pairs reachability problem in one round for networks of diameter 1. In section 4, we show that in two rounds one can compute an approximation for the APSP problem (also for networks of diameter 1). In section 5, we prove lower bounds for computing reachability and distances in networks of diameter 2.

3 All-Pairs Reachability for Networks of Diameter 1

In this section we show that when the diameter of the underlying network is 1, the directed single-source reachability problem can be solved in $O(1)$ rounds in the Broadcast CONGEST model. In fact, we show that it can be solved in a single round. Furthermore, our algorithm can solve the much more general problem of all-pairs reachability (again in a single round). The algorithm is extremely simple. Every vertex simply sends its in-degree and out-degree to all its neighbors in the underlying network, and then, by using this information only, each vertex can determine (by a simple computation) which vertex is reachable from which. This requires messages of at most $2\lceil \log_2 n \rceil$ bits (moreover, if there are no anti-parallel edges then, as the underlying diameter is 1, the in-degree plus out-degree of every vertex is exactly $n - 1$ and therefore it is enough to send only the in-degree and so $\lceil \log_2 n \rceil$ bits are enough).

The next lemma shows that when the underlying diameter of some directed graph H is 1, we can determine if $E(H) \cap (A \times A^c) = \varnothing$ by using the in and out degrees of the vertices in A, for every subset of vertices $A \subseteq V(H)$.

▶ **Lemma 3.1.** *For every directed graph $H = (V, E)$ with underlying diameter 1 and every set $A \subseteq V$, we have $\sum_{v \in A}(d_{in}(v) - d_{out}(v)) = |A^c \times A|$ if and only if $E \cap (A \times A^c) = \varnothing$.*

Proof. Let $H = (V, E)$ be a directed graph with underlying diameter 1 and let A be some subset of V. We have $\sum_{v \in A} d_{out}(v) = |E \cap (A \times V)| = |E \cap (A \times A)| + |E \cap (A \times A^c)|$ and similarly $\sum_{v \in A} d_{in}(v) = |E \cap (V \times A)| = |E \cap (A \times A)| + |E \cap (A^c \times A)|$. It follows that

$$\sum_{v \in A}(d_{in}(v) - d_{out}(v)) = |E \cap (A^c \times A)| - |E \cap (A \times A^c)| \qquad (1)$$

Now, for showing the first direction, assume that $\sum_{v \in A}(d_{in}(v) - d_{out}(v)) = |A^c \times A|$. By equation (1), we have $|A^c \times A| = |E \cap (A^c \times A)| - |E \cap (A \times A^c)|$. As $|E \cap (A^c \times A)| \leq |A^c \times A|$, we get that $|E \cap (A \times A^c)| \leq 0$ and so $E \cap (A \times A^c) = \varnothing$.

For the second direction, assume that $E \cap (A \times A^c) = \varnothing$. Since in addition H has underlying diameter 1, every vertex in A^c must have an outgoing edge to every vertex in A and so $E \cap (A^c \times A) = A^c \times A$. It follows, by equation (1), that $\sum_{v \in A} (d_{in}(v) - d_{out}(v)) = |A^c \times A|$. ◀

The next lemma shows that when the underlying diameter of some directed graph H is 1, the in and out degrees of all the vertices in H are enough to determine which vertices are reachable from any given vertex in H.

▶ **Lemma 3.2.** *For every directed graph H with underlying diameter 1, every ordering $(v_1, ..., v_n)$ of its vertices such that $d_{out}(v_1) \leq ... \leq d_{out}(v_n)$ and every $i \in \{1, ..., n\}$, there exists an index $k \in \{i, ..., n\}$ such that the set of reachable vertices from v_i in H is equal to $\{v_1, ..., v_k\}$. Moreover, k is the minimal index in $\{i, ..., n\}$ for which $(n-k)k = \sum_{j=1}^{k}(d_{in}(v_j) - d_{out}(v_j))$.*

Proof. Let $H = (V, E)$ be a directed graph with underlying diameter 1, let $(v_1, ..., v_n)$ be an ordering of its vertices such that $d_{out}(v_1) \leq ... \leq d_{out}(v_n)$ and let $i \in \{1, ..., n\}$. Let A be the set of all the reachable vertices from v_i in H, and note that we must have $E \cap (A \times A^c) = \varnothing$ (as otherwise v_i can reach a vertex from A^c which is of course contradiction to the definition of A and A^c).

Let k be the highest index in $\{i, ..., n\}$ for which $v_k \in A$ (such an index must exist as $v_i \in A$). Clearly, we have $A \subseteq \{v_1, ..., v_k\}$. We claim that we must also have $\{v_1, ..., v_k\} \subseteq A$. Since $v_k \in A$ and $E \cap (A \times A^c) = \varnothing$, the set A must contain at least $d_{out}(v_k) + 1$ vertices (the vertex v_k and its $d_{out}(v_k)$ outgoing neighbors). It also follows that every vertex in A^c must have out-degree at least $d_{out}(v_k) + 1$. To see this, note that every vertex in A^c must have an outgoing edge to every vertex in A (as $E \cap (A \times A^c) = \varnothing$ and the underlying diameter of H is 1). Therefore, it must be that $v_j \in A$ for all $j \in \{1, ..., k\}$ as $d_{out}(v_j) \leq d_{out}(v_k)$ for every such j. We conclude that $A = \{v_1, ..., v_k\}$.

Now, as $E \cap (A \times A^c) = \varnothing$ we get from Lemma 3.1 that $(n-k)k = \sum_{j=1}^{k}(d_{in}(v_j) - d_{out}(v_j))$. We are left to show that k is the minimal index in $\{i, ..., n\}$ with this property. Assume towards a contradiction that there exists $m \in \{i, ..., n\}$ such that $m < k$ and $\sum_{j=1}^{m}(d_{in}(v_j) - d_{\text{out}}(v_j)) = (n-m)m$. Let $B = \{v_1, ..., v_m\}$. By Lemma 3.1 we get that $E \cap (B \times B^c) = \varnothing$, and, in particular, that v_k is not reachable from v_i (as $v_i \in B$ and $v_k \notin B$) which is a contradiction. ◀

Lemma 3.2 can be easily turned into an algorithm that solves the all-pairs reachability problem in one round (when the diameter of the underlying network is 1) as follows. Each vertex v in the graph starts by broadcasting the values of $d_{in}(v)$ and $d_{out}(v)$. After receiving the messages, v sorts the vertices in non-decreasing order of their out-degree. Let $(v_1, ..., v_n)$ be that ordering. It then finds for every $i \in \{1, ..., n\}$ the minimal index $k_i \in \{i, ..., n\}$ such that $(n-k_i)k_i = \sum_{j=1}^{k_i}(d_{in}(v_j) - d_{out}(v_j))$ and deduces by Lemma 3.2 that the set of reachable vertices from v_i is $\{v_1, ..., v_{k_i}\}$. We conclude the following:

▶ **Corollary 3.3.** *In the Broadcast CONGEST model, there is a deterministic algorithm that solves the all-pairs reachability problem in one round when the diameter of the underlying network is 1.*

We also note that the time complexity of the internal computation of each vertex is $O(n^2)$.

4 APSP Approximation for Networks of Diameter 1

In the previous section, we showed that it is possible to solve the all-pairs reachability problem in one round for networks of diameter 1. Here we show that it is actually possible to compute an approximation to the distance between all pairs of vertices in two rounds.

Let $G = (V, E)$ be a directed graph on n vertices and underlying diameter 1. For every non-negative integer $i < n$, we let $A(i)$ be the set of all the vertices $u \in V$ whose out-degree is greater than i and that have some in-going neighbor whose out-degree is at most i, that is, $A(i) = \{u \mid (d_{out}(u) > i) \text{ and } (\exists w \in V \text{ s.t. } (w, u) \in E \text{ and } d_{out}(w) \leq i)\}$. We also set $M(i)$ to be $\perp$ if $A(i) = \varnothing$ and $\max\{d_{out}(v) \mid v \in A(i)\}$ otherwise.

For each $i \in \{d_{out}(v) \mid v \in V\}$, we set $f_0[i] = i$, and then for each $k \in \{1, ..., n\}$, we further set $f_k[i]$ to be $\perp$ if $f_{k-1}[i] = \perp$ and to $M(f_{k-1}[i])$ otherwise. We first prove some basic properties.

▷ Claim 4.1. For every $v \in V$ and $k \in \{1, ..., n\}$ such that $f_k[d_{out}(v)] \neq \perp$, we have (i) $f_i[d_{out}(v)] \neq \perp$ for every $i \in \{0, ..., k\}$, and (ii) $f_i[d_{out}(v)] < f_{i+1}[d_{out}(v)]$ for every $i \in \{0, ..., k-1\}$.

Proof. The first property follows directly from the definition of the sequence. For the second property, note that if $f_{i+1}[d_{out}(v)] \neq \perp$ holds for some $i \in \{0, ..., k-1\}$, then $f_{i+1}[d_{out}(v)]$ must be equal to the maximum out-degree of the vertices in $A(f_i[d_{out}(v)])$. As, by definition, $A(f_i[d_{out}(v)])$ contains only vertices whose out-degree is greater than $f_i[d_{out}(v)]$, it must be that $f_{i+1}[d_{out}(v)] > f_i[d_{out}(v)]$. ◁

Note that, in particular, this claim implies that $f_n[d_{out}(v)] = \perp$ for every $v \in V$. As otherwise, we would get that $0 \leq f_0[d_{out}(v)] < f_1[d_{out}(v)] < ... < f_n[d_{out}(v)]$, and so that $f_n[d_{out}(v)] \geq n$ which is impossible as the maximum possible out-degree is $n-1$.

▷ Claim 4.2. For every $k \in \{0, ..., n-1\}$ and every two vertices x and y in V such that y is reachable from x, if $f_k[d_{out}(x)] \neq \perp$ and $d_{out}(y) > f_k[d_{out}(x)]$ then $f_{k+1}[d_{out}(x)] \neq \perp$.

Proof. Let x and y be two vertices in V such that y is reachable from x, and let $k \in \{0, ..., n-1\}$ be such that $f_k[d_{out}(x)] \neq \perp$ and $d_{out}(y) > f_k[d_{out}(x)]$. First, note that by definition and Claim 4.1, we must have $d_{out}(x) = f_0[d_{out}(x)] \leq f_k[d_{out}(x)]$. This fact together with the assumption that $d_{out}(y) > f_k[d_{out}(x)]$ implies that G must contain an edge (u, v) such that $d_{out}(u) \leq f_k[d_{out}(x)]$ and $d_{out}(v) > f_k[d_{out}(x)]$. To see this, note that there must be a path π from x to y (as y is reachable from x) and $d_{out}(x) \leq f_k[d_{out}(x)] < d_{out}(y)$. This is possible only if π contains an edge (u, v) such that $f_k[d_{out}(x)] \geq d_{out}(u)$ and $f_k[d_{out}(x)] < d_{out}(v)$. It follows that $M(f_k[d_{out}(x)]) \neq \perp$ and so $f_{k+1}[d_{out}(x)] \neq \perp$. ◁

In the next claims, we show how the defined sequences can be used to estimate the distances between the vertices in the graph.

▷ Claim 4.3. For every two vertices x and y in V such that y is reachable from x, there exists an index $k \in \{0, ..., n-1\}$ such that $f_k[d_{out}(x)] \neq \perp$ and $d_{out}(y) \leq f_k[d_{out}(x)]$. Moreover, if k is the minimal index with this property, then the distance from x to y is at least k.

Proof. Let x and y be two vertices in V such that y is reachable from x, and let π be some shortest path from x to y. Let k' be the maximal index in $\{0, ..., n\}$ for which $f_{k'}[d_{out}(x)] \neq \perp$ (such an index must exist as $f_0[d_{out}(x)] \neq \perp$ always holds). Note that $k' \leq n-1$ as we must have $f_n[d_{out}(v)] = \perp$. It follows that $d_{out}(y) \leq f_{k'}[d_{out}(x)]$ (as if $d_{out}(y) > f_{k'}[d_{out}(x)]$ then, by Claim 4.2, we must have $f_{k'+1}[d_{out}(x)] \neq \perp$ which is a contradiction to the maximality of k') and so the required index exists.

Now, let k be the minimal index in $\{0, ..., n-1\}$ with the required property. We claim that π contains at least $k+1$ different vertices. Indeed, if $k = 0$ then π clearly contains at least 1 vertex (or more if $x \neq y$). This leaves us with the case in which $k > 0$. Note that for every $t \in \{0, ..., k-1\}$, we have $d_{out}(x) = f_0[d_{out}(x)] \leq f_t[d_{out}(x)]$ and $d_{out}(y) > f_t[d_{out}(x)]$. This means that π contains an edge (x'_t, x_t) such that $d_{out}(x'_t) \leq f_t[d_{out}(x)]$ and $d_{out}(x_t) > f_t[d_{out}(x)]$, for every $t \in \{0, ..., k-1\}$. It follows that for every such t, there exists a vertex x_t in π such that $f_{t+1}[d_{out}(x)] \geq d_{out}(x_t) > f_t[d_{out}(x)]$. Note that $d_{out}(x) = f_0[d_{out}(x)] < d_{out}(x_0) < \cdots < d_{out}(x_{k-1})$ which implies that π contains at least $k+1$ different vertices. ◁

▷ Claim 4.4. For every two vertices x and y in V, if $d_{out}(x) \geq d_{out}(y)$ then G contains a path from x to y of length at most 2.

Proof. Let x and y be two vertices in V such that $d_{out}(x) \geq d_{out}(y)$ and assume towards a contradiction that the claim does not hold. We must have $x \neq y$ and $N_{in}(y) \cap (\{x\} \cup N_{out}(x)) = \varnothing$ as otherwise the distance from x to y would be at most 2. Since the underlying diameter of G is 1, we get that $\{x\} \cup N_{out}(x) \subseteq N_{out}(y)$, and so that $d_{out}(y) = |N_{out}(y)| \geq |\{x\} \cup N_{out}(x)| > |N_{out}(x)| = d_{out}(x)$ which is a contradiction. ◁

▷ Claim 4.5. For every $k \in \{0, ..., n-1\}$ and every two vertices x and y in V, if $f_k[d_{out}(x)] \neq \bot$ and $f_k[d_{out}(x)] \geq d_{out}(y)$, then G contains a path from x to y of length at most $3k + 2$.

Proof. Let $k \in \{0, ..., n-1\}$ and $x, y \in V$ be such that $f_k[d_{out}(x)] \neq \bot$ and $f_k[d_{out}(x)] \geq d_{out}(y)$. Let $x_0 = x$ and for each $i \in \{1, ..., k\}$ let x_i and x'_i be two vertices such that $(x'_i, x_i) \in E$ and $d_{out}(x_i) = f_i[d_{out}(x)]$ and $d_{out}(x'_i) \leq f_{i-1}[d_{out}(x)]$ (note that such vertices must exist as $f_i[d_{out}(x)] \neq \bot$ holds for every such i, by Claim 4.1).

We first prove that for each $i \in \{0, ..., k\}$ there exists in G a path from x_0 to x_i of length at most $3i$. We do this by induction on i. The base case ($i = 0$) is trivial. Assume that the claim holds for some $i \in \{0, ..., k-1\}$ and prove it for $i + 1$. We have $d_{out}(x_i) = f_i[d_{out}(x)] \geq d_{out}(x'_{i+1})$ and so, by Claim 4.4, we get that there is a path in G from x_i to x'_{i+1} of length at most 2. As in addition $(x'_{i+1}, x_{i+1}) \in E$, it must be that there is a path of length at most 3 from x_i to x_{i+1} which together with the induction hypothesis gives a path of length at most $3(i + 1)$ from x_0 to x_{i+1}.

Note that we also have $d_{out}(x_k) = f_k[d_{out}(x)] \geq d_{out}(y)$ and so, by Claim 4.4, there is a path of length at most 2 from x_k to y. We conclude that there is a path of length at most $3k + 2$ from x_0 to y. ◁

Now, we put everything together.

▷ Claim 4.6. For every two vertices x and y in V, the following holds:

1. If y is not reachable from x, then there is no index $k \in \{0, ..., n-1\}$ such that $f_k[d_{out}(x)] \neq \bot$ and $d_{out}(y) \leq f_k[d_{out}(x)]$.
2. If y is reachable from x, then such an index exists, and moreover $d(x, y, G) \leq 3k + 2 \leq 3d(x, y, G) + 2$ where k is the minimal index in $\{0, ..., n-1\}$ for which this property holds.

Proof. The first part follows from Claim 4.5 as the existence of such an index would imply the existence of a path from x to y. For the second part, note that by Claim 4.3 such an index exists. Let k be the minimal index with that property. We have $d(x, y, G) \geq k$ (by Claim 4.3) and $d(x, y, G) \leq 3k + 2$ (by Claim 4.5), that is, we have $d(x, y, G) \leq 3k + 2 \leq 3d(x, y, G) + 2$. ◁

The above claim can be easily turned into an algorithm. Each vertex starts by broadcasting its out-degree to all the vertices in the network. In the next round, each vertex finds the maximal out-degree of its outgoing neighbors and broadcasts this value. By using this information, each vertex can compute for every $u \in V$ and $i \in \{0, ..., n\}$ the value of $f_i[d_{out}(u)]$, and then it can compute an estimation for the distance between every pair of vertices x and y by finding the required index as in Claim 4.6.

5 Lower Bounds for Networks of Diameter 2

In this section we prove lower bounds for the single-source reachability problem and the closely related single-source shortest path problem for unweighted directed graphs in the Broadcast CONGEST model that hold even when the underlying network has diameter 2.

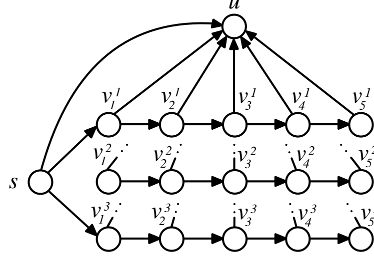


Figure 1 An illustration of the graph $G(k, q, \sigma)$ for $k = 3$, $q = 5$ and $\sigma = 101$.

5.1 The Single-Source Reachability Problem

We start this section by describing a family (parameterized by two positive integers k and q) of directed graphs with underlying diameter at most 2 which we denote by $F_{k,q}$. This family will be used later on to prove the required lower bound.

The Family $F_{k,q}$. For two positive integers $k, q \in \mathbb{Z}$ and a k-bit string $\sigma \in \{0,1\}^k$, we define the directed graph $G(k, q, \sigma)$ to be the graph that consists of:

- k vertex-disjoint directed paths $P_1, ..., P_k$ with q vertices each (that is, $P_i = (V_i, E_i)$ where $V_i = \{v_1^i, ..., v_q^i\}$ and $E_i = \{(v_j^i, v_{j+1}^i) \mid j \in \{1, ..., q-1\}\}$ for every $i \in \{1, ..., k\}$).
- A source vertex s that has an outgoing edge to v_1^i (the first vertex of P_i) if the i-th bit of σ is 1, for every $i \in \{1, ..., k\}$.
- A sink vertex u to which s and every vertex in $P_1, ..., P_k$ has an outgoing edge.

In other words, the vertex set of the graph $G(k, q, \sigma)$ is $V_1 \cup ... \cup V_k \cup \{s, u\}$ and its edge set is $E_1 \cup ... \cup E_k \cup \{(s, v_1^i) \mid i \in \{1, ..., k\} \text{ and } \sigma(i) = 1\} \cup \{(x, u) \mid x \in \{s\} \cup V_1 \cup ... \cup V_k\}$ (see Figure 1 for an illustration). For two positive integers k and q, we define the family $F_{k,q}$ to be the set $\{G(k, q, \sigma) \mid \sigma \in \{0,1\}^k\}$.

Our next goal is to show that any distributed algorithm that solves the single-source reachability problem for all the graphs in $F_{k,q}$ requires a significant number of rounds. We start with the following lemma:

► **Lemma 5.1.** *Let k and q be two positive integers. Let $G \in F_{k,q}$ and let φ be some legal assignment of identifiers to its vertices. Let A be some deterministic distributed algorithm (in the Broadcast CONGEST model) that solves the single-source reachability problem on the instance (G, φ, s) using at most t rounds (for some non-negative integer $t < q$). For each $i \in \{1, ..., k\}$, the output of the vertex v_q^i by the end of the last round is just function of the initial input of the vertices $v_{q-t}^i, ..., v_q^i$ and the sequence of messages that v_q^i received from u.*

Proof. Let $i \in \{1, ..., k\}$. We will show by induction on $0 \le j \le t$ that by the end of the j-th round the state of each vertex $v \in \{v_{q-t+j}^i, ..., v_q^i\}$ is just a function of the initial input of the vertices in its ball of radius j (in the underlying graph of P_i) and the sequence of messages that it received from u up to this round.

The base case ($j = 0$) clearly holds as the state of each vertex in $\{v_{q-t}^i, ..., v_q^i\}$ by the end of round 0 can depend only on its initial input. Assume now that the claim holds for some $0 \le j < t$ and prove it for $j + 1$. Let $r \in \{q - t + j + 1, ..., q\}$. The state of v_r^i by the end of the $(j+1)$-th round is a function of its state at the end of the previous round and the messages that it received from its neighbors in the underlying network (which are u, v_{r-1}^i and possibly v_{r+1}^i).

The messages that v_r^i has received from its neighbors in V_i are, by the induction hypothesis, functions of the inputs of the vertices in the balls of radius j (in P_i) around these neighbors and the sequence of messages that they received from u (up to round j). As u broadcasts the same message to all the vertices in each round, we get that these messages are just a function of the inputs of the vertices in the ball of radius $j+1$ around v_r^i (in P_i) and the sequence of messages that v_r^i received from u (up to round j). As the previous state of v_r^i is, by the induction hypothesis, also a function of the initial inputs of the vertices in its ball of radius j (in P_i) and the sequence of message that it received from u, the claim follows. ◀

▶ **Lemma 5.2.** *Let k and q be two positive integers and let φ be some legal assignment of identifiers to $V_1 \cup ... \cup V_k \cup \{s,u\}$. For every deterministic algorithm A (in the Broadcast CONGEST model), if A solves the single-source reachability problem on all the instances in $\{(G,\varphi,s) \mid G \in F_{k,q}\}$ and uses messages of size at most B bits (for some $B \geq 1$), then A requires at least $\min\{q-1, k/(2B)\}$ rounds.*

Proof. Let A be some deterministic algorithm that satisfies the requirements of the lemma and let t be its running time. We can assume that $t \leq q-2$ as otherwise there is nothing to show.

For each $G \in F_{k,q}$, we let $\text{out}(G)$ be the sequence $(\text{out}(v_q^1, G), ..., \text{out}(v_q^k, G))$ where $\text{out}(v_q^i, G)$ is the output of v_q^i when A is invoked on (G, φ, s), for every $i \in \{1, ..., k\}$. Lemma 5.1 implies that for each $G \in F_{k,q}$ the value of $\text{out}(G)$ is just a function of the initial inputs in (G,φ,s) of the vertices $\bigcup_{i=1}^{k}\{v_{q-t}^i, ..., v_q^i\}$ and the sequence of messages that u had broadcast. Since we assumed that $t \leq q-2$, the initial inputs of these vertices is the same in all $\{(G,\varphi,s) \mid G \in F_{k,q}\}$, and so we can have $\text{out}(G) \neq \text{out}(G')$ for two graphs G and G' in $F_{k,q}$ only if the sequence of messages that u had broadcast in the corresponding invocations was different.

In each round, u may send a message that contains at most B bits, that is, a message with 0 bits, or with 1 bit and so on. Therefore, there are $1+2+...+2^B \leq 2^{2B}$ different messages that u may send in each round. It follows that there are at most 2^{2Bt} possible sequences and so we get that $|\{\text{out}(G) \mid G \in F_{k,q}\}| \leq 2^{2Bt}$. Note also that $|\{\text{out}(G) \mid G \in F_{k,q}\}| = 2^k$ as for each $G \in F_{k,q}$ the output should be different. We conclude that $2^k \leq 2^{2Bt}$ and so $t \geq k/2B$. ◀

▶ **Corollary 5.3.** *In the Broadcast CONGEST model, there is no deterministic algorithm that solves the single-source reachability problem in $o(\sqrt{n/\log n})$ rounds even when the diameter of the underlying network is always 2.*

Proof. Assume towards a contradiction that there exists a deterministic algorithm A that solves the above problem in $T(n) = o(\sqrt{n/\log n})$ rounds. As A works in the CONGEST model, there must be some constant $c \geq 1$ such that the number of bits in any message that the algorithm may send (when it is invoked on inputs of size $n > 1$) is at most $c \cdot \log n$.

Since $T(n) = o(\sqrt{n/\log n})$, there must be some integer $n_0 \geq 16$ for which $T(n) \leq \frac{1}{10c}\sqrt{n/\log n}$ holds for every $n > n_0$. Choose an integer $m > n_0$ such that both $k = \sqrt{m \log m}$ and $q = \sqrt{m/\log m}$ are positive integers. Lemma 5.2 implies that there must be some $G \in F_{k,q}$ and some assignment of identifiers φ to $V(G)$ such that invoking the algorithm on (G,φ,s) requires at least $\min\{\frac{1}{2}q, \frac{1}{4c}\frac{k}{\log m}\} = \min\{\frac{1}{2}\sqrt{\frac{m}{\log m}}, \frac{1}{4c}\sqrt{\frac{m}{\log m}}\} = \frac{1}{4c}\sqrt{\frac{m}{\log m}}$ rounds. But, we also have $|V(G)| > m > n_0$ and so the algorithm must take at most $\frac{1}{5c}\sqrt{m/\log m}$ rounds on (G,φ,s), a contradiction. ◀

In the next lemma, we show that the same lower bound holds for distributed randomized algorithms as well.

▶ **Lemma 5.4.** *Let k and q be two positive integers and let φ be some legal assignment of identifiers to $V_1 \cup ... \cup V_k \cup \{s, u\}$. For every randomized algorithm A (in the Broadcast CONGEST model), if A correctly solves the SSR problem on each instance in $\{(G, \varphi, s) \mid G \in F_{k,q}\}$ with probability $> 1/2$ and uses messages of size at most B bits (for some $B \geq 1$), then A requires at least $\min\{q-1, (k-1)/(2B)\}$ rounds.*

Proof. Clearly, it is sufficient to show that this lower bound holds in a model that generates a public random string first, announces it to every vertex in the graph and then every vertex proceeds deterministically as usual. Let A be a randomized algorithm that works in the above model and solves the SSR problem on every instance in $F = \{(G, \varphi, s) \mid G \in F_{k,q}\}$ with probability $> 1/2$. We can assume that its running time t is at most $q - 2$. As in the proof of Lemma 5.2, we can show that, for every fixed random string r, the algorithm (given that string) can succeed on at most 2^{2Bt} of the instances in F.

For each graph G in $F_{k,q}$, let R_G be the event that the algorithm fails on (G, φ, s). Note that we must have $\sum_{G \in F_{k,q}} P(R_G) \geq |F_{k,q}| - 2^{2Bt}$. By assumption, we must also have $0.5|F_{k,q}| > \sum_{G \in F_{k,q}} P(R_G)$ and so $0.5|F_{k,q}| > |F_{k,q}| - 2^{2Bt}$ which implies that $t > (k-1)/2B$. ◀

5.2 The Single-Source Shortest Path Problem

The result of the previous section already gives a lower bound of $\Omega(\sqrt{n/\log n})$ for the directed SSSP problem (or even for the approximate version of it) as, by definition, any algorithm that solves this problem must also solve the SSR problem.

In this section we show a slightly stronger lower bound of $\Omega(\sqrt{n})$ for this problem which holds even when the diameter of the underlying network is 2 and even when all the vertices in the input graph are guaranteed to be reachable from the given source. As in the previous section, we start by describing a family J_k of unweighted directed graphs with underlying diameter 2 which will be used to prove the lower bound.

The Family J_k. For a positive integer k and a sequence σ of k numbers from $\{1, ..., k\}$, we define the directed graph $G(k, \sigma)$ to be the graph that consists of:

- k vertex-disjoint directed paths $P_1, ..., P_k$ where each $P_i = (V_i, E_i)$ contains $\sigma(i) + k$ vertices. For each path P_i, we denote by u^i its first vertex and by $v^i_1, ..., v^i_k$ its last k vertices.
- A source vertex s that has an outgoing edge to the first vertex of every path in $\{P_1, ..., P_k\}$.
- A sink vertex u to which s and every vertex in $P_1, ..., P_k$ has an outgoing edge.

In other words, the vertex set of the graph $G(k, \sigma)$ is $V_1 \cup ... \cup V_k \cup \{s, u\}$ and its edge set is $E_1 \cup ... \cup E_k \cup \{(s, u^i) \mid i \in \{1, ..., k\}\} \cup \{(x, u) \mid x \in \{s\} \cup V_1 \cup ... \cup V_k\}$ (see Figure 2 for an illustration). For a positive integer k, we define the family J_k to be the set $\{G(k, \sigma) \mid \sigma \in \{1, ..., k\}^k\}$.

We say that a collection of assignments $\{\varphi_G \mid G \in J_k\}$ is a consistent set of assignments for the family J_k, if φ_G is a legal assignment of identifiers to $V(G)$ and $\varphi_G(x) = \varphi_{G'}(x)$, for every $G, G' \in J_k$ and $x \in \{u\} \cup (\bigcup_{i=1}^{k} \{v^i_1, ..., v^i_k\})$.

▶ **Lemma 5.5.** *Let $k > 1$ be some integer and let $\{\varphi_G \mid G \in J_k\}$ be some consistent set of assignments for J_k. For every deterministic algorithm A (in the Broadcast CONGEST model), if A solves the SSSP problem on all the instances in $\{(G, \varphi_G, s) \mid G \in J_k\}$, then A requires $\Omega(k)$ rounds.*

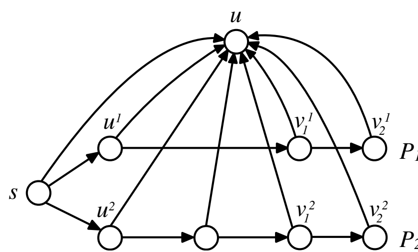


Figure 2 An illustration of the graph $G(k, \sigma)$ for $k = 2$ and $\sigma = (1, 2)$.

Proof. Let A be some deterministic algorithm that satisfies the requirements of the lemma and let t be its running time. We can assume that $t \leq k-2$ as otherwise there is nothing to show. As A works in the CONGEST model, there must be some constant $c \geq 1$ such that the number of bits in any message that the algorithm may send on any input of size $n > 1$ is at most $c \cdot \log n$.

Consider invoking the algorithm A on the instance (G, φ_G, s) for some $G \in J_k$. In each round, u may send one message containing at most $c \cdot \log(|V(G)|) \leq 4c \cdot \log(k)$ bits to all the vertices in the graph. Given that, it is easy to see (by a proof similar to Lemma 5.1) that the output of every v_k^i is just a function of its initial input, the initial input of at most $t \leq k-2$ vertices that precede it in the path P_i, and the sequence of messages that u had broadcast.

For each $G \in J_k$, we let $\text{out}(G)$ be the sequence $(\text{out}(v_k^1, G), ..., \text{out}(v_k^k, G))$ where $\text{out}(v_k^i, G)$ is the output of v_k^i when A is invoked on (G, φ_G, s), for every $i \in \{1, ..., k\}$. By the observation above, for each $i \in \{1, ..., k\}$ the vertices in P_i whose initial input may affect the output of v_k^i are just $v_2^i, ..., v_k^i$. Since the initial input of each of these vertices is the same in each of the instances in $\{(G, \varphi_G, s) \mid G \in J_k\}$, we get that $\text{out}(G) \neq \text{out}(G')$ can hold for some graphs G and G' in J_k only if the sequence of messages that u had broadcast in the corresponding invocations was different.

Straightforward calculations show that the number of such sequences is at most $(2^{8c \cdot \log(k)})^t = k^{8c \cdot t}$, and so $|\{\text{out}(G) \mid G \in J_k\}| \leq k^{8c \cdot t}$. Since we assumed that the algorithm is correct, we must have $\text{out}(G) \neq \text{out}(G')$ for every two different graphs G and G' in J_k (as, by construction, we cannot have $d(s, v_k^i, G) = d(s, v_k^i, G')$ for every $i \in \{1, ..., k\}$), and so $|\{\text{out}(G) \mid G \in J_k\}| = |J_k| = k^k$. We conclude that $k^k \leq k^{8c \cdot t}$ and so $t \geq k/(8c)$. ◀

▶ **Corollary 5.6.** *In the Broadcast CONGEST model, there is no deterministic algorithm that solves the SSSP problem in $o(\sqrt{n})$ rounds even when the diameter of the underlying network is always 2.*

Proof. Assume towards a contradiction that there exists a deterministic algorithm A that solves the above problem in $T(n) = o(\sqrt{n})$ rounds. As before, we can assume that there is a constant $c \geq 1$ such that the number of bits in any message that A may send on any input of size $n > 1$ is at most $c \cdot \log n$.

Since $T(n) = o(\sqrt{n})$, there must be some positive integer n_0 for which $T(n) \leq \frac{1}{18c}\sqrt{n}$ holds for every $n > n_0$. Let $k > 1$ be an integer such that $k^2 > n_0$. The proof of Lemma 5.5 implies that there must be some $G \in J_k$ and some assignment of identifiers φ to $V(G)$ such that invoking the algorithm on (G, φ, s) requires at least $k/(8c)$ rounds. But, $|V(G)| > k^2 > n_0$ and so the algorithm must take at most $\frac{1}{18c}\sqrt{|V(G)|} \leq \frac{1}{9c}k$ rounds on (G, φ, s), a contradiction. ◀

By a proof similar to that of the previous section, it is possible to show that the same lower bound holds for randomized distributed algorithms as well.

References

1. Udit Agarwal, Vijaya Ramachandran, Valerie King, and Matteo Pontecorvi. A Deterministic Distributed Algorithm for Exact Weighted All-Pairs Shortest Paths in $\tilde{O}(n^{3/2})$ Rounds. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing (PODC '18)*, pages 199–205, 2018.
2. Ruben Becker, Andreas Karrenbauer, Sebastian Krinninger, and Christoph Lenzen. Near-Optimal Approximate Shortest Paths and Transshipment in Distributed and Streaming Models. In *Proceedings of the 31st International Symposium on Distributed Computing (DISC '17)*, pages 7:1–7:16, 2017.
3. Aaron Bernstein and Danupon Nanongkai. Distributed Exact Weighted All-pairs Shortest Paths in Near-linear Time. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing (STOC '19)*, pages 334–342, 2019.
4. Keren Censor-Hillel, Michal Dory, Janne H. Korhonen, and Dean Leitersdorf. Fast Approximate Shortest Paths in the Congested Clique. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing (PODC '19)*, pages 74–83, 2019.
5. Keren Censor-Hillel, Petteri Kaski, Janne H. Korhonen, Christoph Lenzen, Ami Paz, and Jukka Suomela. Algebraic Methods in the Congested Clique. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing (PODC '15)*, pages 143–152, 2015.
6. Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. Distributed Verification and Hardness of Distributed Approximation. In *Proceedings of the Forty-third Annual ACM Symposium on Theory of Computing (STOC '11)*, pages 363–372, 2011.
7. Michael Elkin. Distributed Exact Shortest Paths in Sublinear Time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing (STOC '17)*, pages 757–770, 2017.
8. Sebastian Forster and Danupon Nanongkai. A Faster Distributed Single-Source Shortest Paths Algorithm. In *Proceedings of the 59th Annual IEEE Symposium on Foundations of Computer Science (FOCS '18)*, pages 686–697, 2018.
9. Mohsen Ghaffari and Jason Li. Improved Distributed Algorithms for Exact Shortest Paths. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing (STOC'18)*, pages 431–444, 2018.
10. Mohsen Ghaffari and Rajan Udwani. Brief Announcement: Distributed Single-Source Reachability. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing (PODC '15)*, pages 163–165, 2015.
11. Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. A Deterministic Almost-tight Distributed Algorithm for Approximating Single-source Shortest Paths. In *Proceedings of the Forty-eighth Annual ACM Symposium on Theory of Computing (STOC '16)*, pages 489–498, 2016.
12. Stephan Holzer and Roger Wattenhofer. Optimal Distributed All Pairs Shortest Paths and Applications. In *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing (PODC '12)*, pages 355–364, 2012.
13. Chien-Chung Huang, Danupon Nanongkai, and Thatchaphol Saranurak. Distributed Exact Weighted All-Pairs Shortest Paths in $\tilde{O}(n^{5/4})$ Rounds. In *Proceedings of the 58th Annual IEEE Symposium on Foundations of Computer Science (FOCS '17)*, pages 168–179, 2017.
14. Christoph Lenzen and Boaz Patt-Shamir. Fast Partial Distance Estimation and Applications. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing (PODC '15)*, pages 153–162, 2015.
15. Christoph Lenzen and David Peleg. Efficient Distributed Source Detection with Limited Bandwidth. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing (PODC '13)*, pages 375–382, 2013.

16 Danupon Nanongkai. Distributed Approximation Algorithms for Weighted Shortest Paths. In *Proceedings of the Forty-sixth Annual ACM Symposium on Theory of Computing (STOC '14)*, pages 565–573, 2014.

17 David Peleg. *Distributed Computing: A Locality-sensitive Approach.* Society for Industrial and Applied Mathematics, 2000.

On the Round Complexity of Randomized Byzantine Agreement

Ran Cohen
Boston University, MA, USA
Northeastern University, Boston, MA, USA
rancohen@ccs.neu.edu

Iftach Haitner
School of Computer Science, Tel Aviv University, Israel
iftachh@cs.tau.ac.il

Nikolaos Makriyannis
Department of Computer Science, Technion, Haifa, Israel
n.makriyannis@gmail.com

Matan Orland
School of Computer Science, Tel Aviv University, Israel
matanorland@mail.tau.ac.il

Alex Samorodnitsky
School of Engineering and Computer Science, The Hebrew University of Jerusalem, Israel
salex@cs.huji.ac.il

Abstract

We prove lower bounds on the round complexity of *randomized* Byzantine agreement (BA) protocols, bounding the halting probability of such protocols after one and two rounds. In particular, we prove that:

1. BA protocols resilient against $n/3$ [resp., $n/4$] corruptions terminate (under attack) at the end of the first round with probability at most $o(1)$ [resp., $1/2 + o(1)$].
2. BA protocols resilient against $n/4$ corruptions terminate at the end of the second round with probability at most $1 - \Theta(1)$.
3. For a large class of protocols (including all BA protocols used in practice) and under a plausible combinatorial conjecture, BA protocols resilient against $n/3$ [resp., $n/4$] corruptions terminate at the end of the second round with probability at most $o(1)$ [resp., $1/2 + o(1)$].

The above bounds hold even when the parties use a trusted setup phase, e.g., a public-key infrastructure (PKI).

The third bound essentially matches the recent protocol of Micali (ITCS'17) that tolerates up to $n/3$ corruptions and terminates at the end of the third round with constant probability.

2012 ACM Subject Classification Theory of computation → Cryptographic protocols

Keywords and phrases Byzantine agreement, lower bound, round complexity

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.12

Related Version A full version of the paper is available at `https://arxiv.org/abs/1907.11329`.

Funding *Ran Cohen*: Research supported by the Northeastern University Cybersecurity and Privacy Institute Post-doctoral fellowship, IARPA under award 2019-19020700009 (ACHILLES), NSF grant TWC-1664445, NSF grant 1422965, and by the NSF MACS project. Some of this work was done while the author was a post-doc at Tel Aviv University, supported by ERC starting grant 638121.
Iftach Haitner: Member of the Check Point Institute for Information Security. Research supported by ERC starting grant 638121.
Nikolaos Makriyannis: Research supported by ERC starting grant 638121 and by ERC advanced grant 742754.
Matan Orland: Research supported by ERC starting grant 638121.
Alex Samorodnitsky: Research partially supported by ISF grant 1724/15.

© Ran Cohen, Iftach Haitner, Nikolaos Makriyannis, Matan Orland, and Alex Samorodnitsky;
licensed under Creative Commons License CC-BY

Acknowledgements We would like to thank Rotem Oshman, Juan Garay, Ehud Friedgut, and Elchanan Mossel for very helpful discussions.

1 Introduction

Byzantine agreement (BA) [56, 43] is one of the most important problems in theoretical computer science. In a BA protocol, a set of n parties wish to jointly agree on one of the honest parties' input bits. The protocol is *t-resilient* if no set of t corrupted parties can collude and prevent the honest parties from completing this task. In the closely related problem of *broadcast*, all honest parties must agree on the message sent by a (potentially corrupted) sender. Byzantine agreement and broadcast are fundamental building blocks in distributed computing and cryptography, with applications in fault-tolerant distributed systems [14, 42], secure multiparty computation [60, 33, 7, 15], and more recently, blockchain protocols [16, 31, 55].

In this work, we consider the *synchronous* communication model, where the protocol proceeds in rounds. It is well known that in the plain model, without any trusted setup assumptions, BA and broadcast can be solved if and only if $t < n/3$ [56, 43, 26, 30]. Assuming the existence of digital signatures and a public-key infrastructure (PKI), BA can be solved in the honest-majority setting $t < n/2$, and broadcast under any number of corruptions $t < n$ [23]. Information-theoretic variants that remain secure against computationally unbounded adversaries exist using information-theoretic pseudo-signatures [57].

An important aspect of BA and broadcast protocols is their *round complexity*. Deterministic t-resilient protocols require at least $t + 1$ rounds [25, 23], which is a tight lower bound [23, 30]. The breakthrough results of Ben-Or [6] and Rabin [58] showed that this limitation can be circumvented using randomization. In particular, Rabin [58] used *random beacons* (common random coins that are secret-shared among the parties in a trusted setup phase) to construct a BA protocol resilient to $t < n/4$ corruptions. Rabin's protocol fails with probability 2^{-r} after r rounds, and requires *expected* constant number of rounds to reach agreement. This line of research culminated with the work of Feldman and Micali [24] who showed how to compute the common coins from scratch, yielding expected-constant-round BA protocol in the plain model, resilient to $t < n/3$ corruptions. Katz and Koo [40] gave an analogue result in the PKI-model for the honest-majority case. Recent results used trusted setup and cryptographic assumptions to establish a surprisingly small expected round complexity, namely 9 for $t < n/3$ [47] and 10 for $t < n/2$ [49, 2].

The expected-constant-round protocols mentioned above are guaranteed to terminate (with negligible error probability) within a poly-logarithmic number of rounds. The lower bounds on the guaranteed termination from [25, 23] were generalized by [18, 39], showing that any randomized r-round protocol must fail with probability at least $(c \cdot r)^{-r}$ for some constant c. However, to date there is no lower bound on the *expected* round complexity of randomized BA.

In this work, we tackle this question and show new lower bounds for randomized BA. To make the discussion more informative, we consider a more explicit definition that bounds the halting probability within a specific number of rounds. A lower bound based on such a definition readily implies a lower bound on the expected round complexity of the BA protocol.

1.1 The Model

We start with describing in more details the model in which our lower bounds are given. In the BA protocols considered in this work, the parties are communicating over a synchronous network of private and authenticated channels. Each party starts the protocol with an input bit and upon completion decides on an output bit. The protocol is t-resilient if when facing t colluding parties that attack the protocol it holds that: (1) all honest parties agree on the same output bit (*agreement*), and (2) if all honest parties start with the same input bit, then this is the common output bit (*validity*). The protocols might have a *trusted setup phase*: a trusted external party samples correlated values and distributes them between the parties. A setup phase is known to be essential for tolerating $t \geq n/3$ corruptions, and seems to be crucial for highly efficient protocols such as [47, 16, 49, 2, 1]. The trusted setup phase is typically implemented using (heavy) secure multiparty computation [10, 12], via a public-key infrastructure, or with a random oracle (that can be used to model proof of work) [54].

Locally consistent adversaries. The attacks presented in the paper require very limited capabilities from the corrupted parties (a limitation that makes our bounds stronger). Specifically, a corrupted party might (1) prematurely abort, and (2) send messages to different parties based on *differing* input bits and/or incoming messages from other corrupted parties. We emphasize that corrupted parties sample their random coins honestly (and use the same coins for all messages sent). In addition, they do not lie about messages received from honest parties.

Public-randomness protocols. In many randomized protocols, including all those used in practice, cryptography is merely used to provide *message authentication* – preventing a party from lying about the messages it received – and *verifiable randomness* – forcing the parties to toss their coins correctly. The description of such protocols can be greatly simplified if only security against locally consistent adversaries is required (in which corrupted parties do not lie about their coin tosses and their incoming messages from honest parties). This motivates the definition of *public-randomness* protocols, where each party publishes its local coin tosses for each round (the party's first message also contains its setup parameter, if such exists). Although our attacks apply to arbitrary BA protocols, we show even stronger lower bounds for public-randomness protocols.

We illustrate the simplicity of the model by considering the BA protocol of Micali [47]. In this protocol, the cryptographic tools, digital signatures and verifiable random functions (VRFs),[1] are used to allow the parties elect leaders and toss coins with probability 2/3 as follows: each party P_i in round r evaluates the VRF on the pair (i, r) and multicasts the result. The leader is set to be the party with the smallest VRF value, and the coin is set to be the least-significant bit of this value. Since these values are uniformly distributed κ-bit strings (κ is the security parameter), and there are at least $2n/3$ honest parties, the success probability is 2/3. (Indeed, with probability 1/3, the leader is corrupted, and can send its value only to a subset of the parties, creating disagreement.)

When considering locally consistent adversaries, Micali's protocol can be significantly simplified by having each party randomly sample and multicast a uniformly distributed κ-bit string (cryptographic tools and setup phase are no longer needed). Corrupted parties can still send their values to a subset of honest parties as before, but they cannot send different random values to different honest parties.

[1] A pseudorandom function that provides a non-interactively verifiable proof for the correctness of its output.

A similar simplification applies to other BA protocols that are based on leader election and coin tosses such as [24, 27, 40] (private channels are used for a leader-election sub-protocol), [49, 2] (cryptography is used for coin-tossing and message-authentication), and [16, 1] (cryptography is used to elect a small committee per round).[2]

▶ **Proposition 1** (Malicious security to locally consistent public-randomness protocol, informal). *Each of the BA protocols of [24, 27, 40, 47, 16, 49, 2, 1] induces a public-randomness BA protocol secure against locally consistent adversaries, with the same parameters.*

A useful abstraction for protocol design. To complete the picture, we remark that security against locally consistent adversaries, which may seem somewhat weak at first sight, can be compiled using standard cryptographic techniques into security against arbitrary adversaries. This reduction becomes lossless, efficiency-wise and security-wise, when applied to public-randomness protocols. Thus, building public-randomness protocols secure against locally consistent adversaries is a useful abstraction for protocol designers that want to use what cryptography has to offer, but without being bothered with the technical details.

Connection to the full-information model. The public-randomness model can be viewed as a restricted form of the *full-information model* [17, 8, 32, 5, 9, 35, 38, 44, 41, 45]. In the latter model, the adversary is computationally unbounded and has complete access to all the information in the system, i.e., it can listen to all transmitted messages and view the internal states of honest parties (such an adversary is also called *intrusive* [17]). One of the motivations to study full-information protocols is to separate *randomization* from *cryptography* and see to what extent randomization alone can speed up Byzantine agreement. Bar-Joseph and Ben-Or [5] showed that any full-information BA protocol tolerating $t = \Theta(n)$ adaptive, fail-stop corruptions (i.e., the adversary can dynamically choose which parties to crash) runs for $\tilde{\Omega}(\sqrt{n})$ rounds. Goldwasser et al. [35] constructed an $O(\log n)$-round BA protocol tolerating $t = (1/3 - \varepsilon)n$ static, malicious corruptions, for an arbitrarily small constant $\varepsilon > 0$.

We chose to state our results in the public-randomness model for two reasons. First, our lower bounds readily extend to lower bounds in the full-information model (since we consider weaker adversarial capabilities, e.g., all our attacks are efficient). Second, when considering locally consistent adversaries, public-randomness captures essentially what efficient cryptography has to offer. Indeed, all protocol used in practice can be cast as public-randomness protocols tolerating locally consistent adversaries (Proposition 1) and every public-randomness protocol secure against locally consistent adversaries can be compiled, using cryptography, to malicious security in the standard model, where security relies on secret coins (see Theorem 6 below).

We note that it is known how to compile certain full-information protocols and "boost" their security from fail-stop into malicious; however, these compilers capture either deterministic protocols [36, 13, 52] or protocols with a non-uniform source of randomness (namely, an SV-source [59]) [35]. It is unclear whether these compilers can be extended to capture arbitrary protocols (this is in fact stated as an open question in [13, 35]). In addition, these compilers are designed to be information theoretic and not rely on cryptography; thus, they do not model highly efficient protocols used in practice.

[2] Unlike the aforementioned protocols that use "simple" preprocess and "light-weight" cryptographic tools, the protocol of Rabin [58] uses a heavy, per execution, setup phase (consisting of Shamir sharing of a random coin for every potential round) that we do not know how to cast as a public-randomness protocol.

1.2 Our Results

We present three lower bounds on the halting probability of randomized BA protocols. To keep the following introductory discussion simple, we will assume that both validity and agreement properties hold perfectly, without error.

First-round halting. Our first result bounds the halting probability after a single communication round. This is the simplest case since parties cannot inform each other about inconsistencies they encounter. Indeed, the established lower bound is quite strong, showing an exponentially small bound on the halting probability when $t \geq n/3$, and exponentially close to $1/2$ when $t \geq n/4$.

▶ **Theorem 2** (First-round halting, informal)**.** *Let Π be an n-party BA protocol and let γ denote the halting probability after a single communication round facing a locally consistent, static, adversary corrupting t parties. Then,*

- *$t \geq n/3$ implies $\gamma \leq 2^{t-n}$ for arbitrary protocols, and $\gamma = 0$ for public-randomness protocols.*
- *$t \geq n/4$ implies $\gamma \leq 1/2 + 2^{t-n}$ for arbitrary protocols, and $\gamma \leq 1/2$ for public-randomness protocols.*

Note that the deterministic $(t+1)$-round, t-resilient BA protocol of Dolev and Strong [23] can be cast as a locally consistent public-randomness protocol (in the plain model).[3] Theorem 2 shows that for $n = 3$ and $t = 1$, this two-round BA protocol is essentially optimal and cannot be improved via randomization (at least without considering complex protocols that cannot be cast as public-randomness protocols).

Second-round halting for arbitrary protocols. Our second result considers the halting probability after two communication rounds. This is a much more challenging regime, as honest parties have time to detect inconsistencies in first-round messages. Our bound for arbitrary protocols in this case is weaker, and shows that when $t > n/4$, the halting probability is bounded away from 1.

▶ **Theorem 3** (Second-round halting, arbitrary protocols, informal)**.** *Let Π be an n-party BA protocol and let γ denote the halting probability after two communication rounds facing a locally consistent, static, adversary corrupting $t = (1/4 + \varepsilon) \cdot n$ parties. Then, $\gamma \leq 1 - (\varepsilon/5)^2$.*

Second-round halting for public-randomness protocols. Theorem 3 bounds the second-round halting probability of arbitrary BA protocols away from one. For public-randomness protocol we achieve a much stronger bound. The attack requires *adaptive* corruptions (as opposed to *static* corruptions in the previous case) and is based on a combinatorial conjecture that is stated below.[4]

---

[3] When considering locally consistent adversaries, the impossibility of BA for $t \geq n/3$ does not apply.

[4] The attack holds even without assuming Conjecture 5 when considering *strongly adaptive* corruptions [34], in which an adversary sees all messages sent by honest parties in any given round and, based on the messages' content, decides whether to corrupt a party (and alter its message or sabotage its delivery) or not. Similarly, the conjecture is not required if each party is limited to tossing a single unbiased coin. These extensions are not formally proved in this paper.

▶ **Theorem 4** (Second-round halting, public-randomness protocols, informal). *Let* Π *be an* n*-party public-randomness BA protocol and let* γ *denote the halting probability after two communication rounds facing a locally consistent adversary adaptively corrupting* t *parties. Then, for sufficiently large* n *and assuming Conjecture 5 holds,*

- $t > n/3$ *implies* $\gamma = 0$.
- $t > n/4$ *implies* $\gamma \leq 1/2$.

Theorem 4 shows that for sufficiently large n, any public-randomness protocol tolerating $t > n/3$ locally consistent corruptions cannot halt in less than three rounds (unless Conjecture 5 is false). In particular, its expected round complexity must be at least three.

To understand the meaning of this result, recall the protocol of Micali [47]. As discussed above, this protocol can be cast as a public-randomness protocol tolerating $t < n/3$ adaptive locally consistent corruptions. The protocol proceeds by continuously running a three-round sub-protocol until halting, where each sub-protocol consists of a coin-tossing round, a check-halting-on-0 round, and a check-halting-on-1 round. Executing a single instance of this sub-protocol demonstrates a halting probability of 1/3 after three rounds. By Theorem 4, a protocol that tolerates slightly more corruptions, i.e., $(1/3 + \varepsilon) \cdot n$, for arbitrarily small $\varepsilon > 0$, cannot halt in fewer rounds.

Our techniques. Our attacks follow the spirit of many lower bounds on the round complexity on BA and broadcast [25, 23, 39, 22, 29, 4]. The underlying idea is to start with a configuration in which validity assures the common output is 0, and gradually adjust it, while retaining the same output value, into a configuration in which validity assures the common output is 1. (For the simple case of deterministic protocols, each step of the argument requires the corrupted parties to lie about their input bits and incoming messages from other corrupted parties, but otherwise behave honestly.) Our main contribution, which departs from the aforementioned paradigm, is adding another dimension to the attack by aborting a random subset of parties (rather than simply manipulating the input and incoming messages). This change allows us to bypass a seemingly inherent barrier for this approach. We refer the reader to Section 2 for a detailed overview of our attacks.

We remark that a similar approach was employed by Attiya and Censor [3] for obtaining lower bounds on consensus protocols in the asynchronous shared-memory model, a flavor of BA in a communication model very different to the one considered in the present paper. Specifically, [3] showed that in an asynchronous shared-memory system, $\Theta(n^2)$ steps are required for n processors to reach agreement when facing $\Theta(n)$ *computationally unbounded strongly adaptive* corruptions (see Footnote 4). Their adversary also aborts a subset of the parties to prevent halting; however, the difference in communication model (synchronous in our work, vs. asynchronous in [3]) and the adversary's power (efficient and adaptive in our work, vs. computationally unbounded and strongly adaptive in [3]) yields a very different attack and analysis (though, interestingly, both attacks boil down to different variants of isoperimetric-type inequalities).

The combinatorial conjecture. We conclude the present section by motivating and stating the combinatorial conjecture assumed in Theorem 4, and discussing its plausibility. We believe the conjecture to be of independent interest, as it relates to topics from Boolean functions analysis such as influences of subsets of variables [53] and isoperimetric-type inequalities [50, 51]. The nature of our conjecture makes the following paragraphs somewhat technical, and reading them can be postponed until after going over the description of our attack in Section 2.

The analysis of our attack naturally gives rise to an isoperimetric-type inequality. For limited types of protocols, we manage to prove it using Friedgut's theorem [28] about approximate juntas and the KKL theorem [37]. For arbitrary protocols, however, we only manage to reduce our attack to the conjecture below.

We require the following notation before stating the conjecture. Let Σ denote some finite set. For $\boldsymbol{x} \in \Sigma^n$ and $\mathcal{S} \subseteq [n]$, define the vector $\perp_{\mathcal{S}}(\boldsymbol{x}) \in \{\Sigma \cup \perp\}^n$ by assigning all entries indexed by $\mathcal{S}$ with the value $\perp$, and all other entries according to $\boldsymbol{x}$. Finally, let $\mathbf{D}_{n,\sigma}$ denote the distribution induced over subsets of $[n]$ by choosing each element with probability σ independently at random.

▶ **Conjecture 5.** *For any $\sigma, \lambda > 0$ there exists $\delta > 0$ such that the following holds for large enough $n \in \mathbb{N}$: let Σ be a finite alphabet, and let $\mathcal{A}_0, \mathcal{A}_1 \subseteq \{\Sigma \cup \perp\}^n$ be two sets such that for both $b \in \{0,1\}$:*

$$\Pr_{\mathcal{S} \leftarrow \mathbf{D}_{n,\sigma}} \left[\Pr_{\boldsymbol{r} \leftarrow \Sigma^n} [\boldsymbol{r}, \perp_{\mathcal{S}}(\boldsymbol{r}) \in \mathcal{A}_b] \geq \lambda \right] \geq 1 - \delta.$$

Then,

$$\Pr_{\substack{\mathcal{S} \leftarrow \mathbf{D}_{n,\sigma} \\ \boldsymbol{r} \leftarrow \Sigma^n}} [\forall b \in \{0,1\} \colon \{\boldsymbol{r}, \perp_{\mathcal{S}}(\boldsymbol{r})\} \cap \mathcal{A}_b \neq \emptyset] \geq \delta.$$

Consider two large sets $\mathcal{A}_0$ and $\mathcal{A}_1$ which are "stable" in the following sense: for both $b \in \{0,1\}$, with probability $1 - \delta$ over $\mathcal{S} \leftarrow \mathbf{D}_{n,\sigma}$, it holds that both $\boldsymbol{r}$ and $\perp_{\mathcal{S}}(\boldsymbol{r})$ belong to $\mathcal{A}_b$, with probability at least λ over $\boldsymbol{r}$. Conjecture 5 stipulates that with high probability ($\geq \delta$), the vectors $\boldsymbol{r}$ and $\perp_{\mathcal{S}}(\boldsymbol{r})$ lie in opposite sets (i.e., one is in $\mathcal{A}_0$ and the other $\mathcal{A}_{1-b}$), for random $\boldsymbol{r}$ and $\mathcal{S}$. It is somewhat reminiscent of the following flavor of isoperimetric inequality: for any two large sets $\mathcal{B}_0$ and $\mathcal{B}_1$, taking a random element from $\mathcal{B}_0$ and resampling a few coordinates, yields an element in $\mathcal{B}_1$ with large probability. Less formally, one can "move" from one set to the other by manipulating a few coordinates [50, 51].

A few remarks are in order. First, it suffices for our purposes to show that δ is a noticeable (i.e., inverse polynomial) function of n, rather than independent of n.[5] We opted for the latter as it gives a stronger attack. Second, the conjecture holds for "natural" sets such as balls, i.e., $\mathcal{A}_0$ and $\mathcal{A}_1$ are balls centered around 0^n and 1^n of constant radius,[6] and "prefix" sets, i.e., sets of the form $\mathcal{A}_b = b^k \times \{\Sigma \cup \perp\}^{n-k}$. Furthermore, the claim can be proven when the probabilities over $\mathcal{S}$ and $\boldsymbol{r}$ are reversed, i.e., "with probability λ over $\boldsymbol{r}$, it holds that both $\boldsymbol{r}$ and $\perp_{\mathcal{S}}(\boldsymbol{r})$ belong to $\mathcal{A}_b$ with probability at least $1 - \delta$ over $\mathcal{S}$", instead of the above. Interestingly, this weaker statement boils down to the aforementioned isoperimetric-type inequality (c.f. [50] for the Boolean case and [51] for the non Boolean case).

We conclude by pointing out that, as mentioned in Footnote 4, the conjecture is not needed for certain limited cases that are not addressed in detail in the present paper. One such case is sketched out in Section 2.

1.3 Locally Consistent Security to Malicious Security

As briefly mentioned in Section 1.1, protocols that are secure against locally consistent adversaries can be compiled to tolerate arbitrary malicious adversaries. The compiler requires a PKI for digital signatures and verifiable random functions (VRFs) [48]. A VRF is a pseudorandom function with an additional property: using the secret key and an input x,

[5] We remark that it is rather easy to show that $\delta \geq 2^{-n}$, which is not good enough for our purposes.
[6] The alphabet Σ is not necessarily Boolean, and there are a couple of subtleties in defining balls.

the VRF outputs a pseudorandom value y along with a proof string π; using the public key, everyone can use π to verify whether y is the output of x. We consider a trusted setup phase for establishing the PKI, where every party generates keys for a VRF and for a signature scheme, and publishes the corresponding public keys.

Given a protocol that is secure against locally consistent adversaries, the compiled protocol proceeds as follows, round by round. Each party P_i sets its random coins for the r'th round ρ_i^r (together with a proof π_i^r) by evaluating the VRF over the pair (i, r). Next, for every $j \in [n]$, party P_i uses these coins to compute the message $m_{i\to j}^r$ for P_j, signs $m_{i\to j}^r$ along with the VRF proof π_i^r as $\sigma_{i\to j}^r$, and sends $(m_{i\to j}^r, \pi_i^r, \sigma_{i\to j}^r)$ to P_j. Finally, P_i proves to each P_j using a zero-knowledge proof of knowledge that:

1. There exist an input bit b, random coins ρ_i^r, as well as random coins and incoming messages $\rho_i^{r'}$ and $(m_{1\to i}^{r'}, \ldots, m_{n\to i}^{r'})$ for every $r' < r$, such that: (1) π_i^r verifies that ρ_i^r is the VRF output of (i, r) (using the VRF public key of P_i), and (2) the message $m_{i\to j}^r$ is the output of the next-message function of P_i when applied to these values.
2. For every $r' < r$, the input bit b and the random coins $\rho_i^{r''}$ and incoming messages $(m_{1\to i}^{r''}, \ldots, m_{n\to i}^{r''})$ for every $r'' < r'$, are the same as those used to generate $m_{i\to j}^{r'}$.
3. For $r > 1$, the messages received in the previous round are properly signed. That is, for every $k \in [n]$, there is a signature $\sigma_{k\to i}^{r-1}$ of the message $m_{k\to i}^{r-1}$ that verifies under the signature-verification key of P_k.

When considering public-randomness protocols, the above compilation can be made much more efficient. Instead of proving in zero knowledge the consistency of each message, each party P_i concatenates to each message all of its incoming messages from the previous round. A receiver can now locally verify the coins used by P_i are the VRF output of (i, r) (as assured by the VRF), that the incoming messages are properly signed, and that the message is correctly generated from the internal state of P_i (which is now visible and verified).

▶ **Theorem 6** (Locally consistent to malicious security, folklore, informal). *Assume PKI for digital signatures and VRF. Then, a BA protocol secure against locally consistent adversaries can be compiled into a maliciously secure BA protocol with the same parameters, apart from a constant blowup in the round complexity (no blowup for public-randomness protocols).*

1.4 Additional Related Work

Following the work of Feldman and Micali [24] in the two-thirds majority setting, Katz and Koo [40] improved the expected round complexity to 23, and Micali [47] to 9. In the honest-majority setting, Fitzi and Garay [27] showed expected-constant-round protocol and Katz and Koo [40] expected 56 rounds. Micali and Vaikuntanathan [49] adjusted the technique from [47] to the honest-majority case. Abraham et al. [2] achieved expected 10 rounds assuming static corruptions and expected 16 rounds assuming adaptive corruptions. Abraham et al. [1] constructed an expected-constant-round protocol tolerating $(1/2 - \epsilon) \cdot n$ adaptive corruptions with sublinear communication complexity. In the dishonest-majority setting, Garay et al. [29] constructed a broadcast protocol with expected $O(k)$ rounds, tolerating $t < n/2 + k$ corruptions.

Attiya and Censor-Hillel [4] extended the results of Chor et al. [18] and of Karlin and Yao [39] on guaranteed termination of randomized BA protocols to the asynchronous setting, and provided a tight lower bound.

Randomized protocols with expected constant round complexity have *probabilistic termination*, which requires delicate care with respect to composition (i.e., their usage as subroutines by higher-level protocols). Parallel composition of randomized BA protocols was analyzed in [6, 27], sequential composition in [46], and universal composition in [20, 19].

1.5 Open Questions

Our attack on two-round halting of public-randomness protocols is based on Conjecture 5. In this work we prove special cases of this conjecture, but proving the general case remains an open challenge.

A different interesting direction is to bound the halting probability of protocols when $t < n/4$. It is not clear how to extend our attacks to this regime.

2 Technical Overview

In this section, we outline our techniques for proving our results; we refer the reader to the full version of the paper [21] for formal claims and complete proofs. We start with explaining our bound for first-round halting of arbitrary protocols (Theorem 2). We then move to second-round halting, starting with the weaker bound for arbitrary protocols (Theorem 3), and then move to the much stronger bound for public-randomness protocols (Theorem 4).

Notations. We use calligraphic letters to denote sets, uppercase for random variables, lowercase for values, boldface for vectors, and sans-serif (e.g., A) for algorithms (i.e., Turing Machines). For $n \in \mathbb{N}$, let $[n] = \{1, \cdots, n\}$ and $(n) = \{0, 1, \cdots, n\}$. Let $\operatorname{dist}(x, y)$ denote the hamming distance between x and y. For a set $\mathcal{S} \subseteq [n]$ let $\overline{\mathcal{S}} = [n] \setminus \mathcal{S}$. For a set $\mathcal{R} \subseteq \{0,1\}^n$, let $\mathcal{R}|_{\mathcal{S}} = \{\boldsymbol{x}_{\mathcal{S}} \in \{0,1\}^{|\mathcal{S}|} \text{ s.t. } \boldsymbol{x} \in \mathcal{R}\}$, i.e., $\mathcal{R}|_{\mathcal{S}}$ is the projection of $\mathcal{R}$ on the index-set $\mathcal{S}$.

Fix an n-party randomized BA protocol $\Pi = (\mathsf{P}_1, \ldots, \mathsf{P}_n)$. For presentation purposes, we assume that validity and agreement hold *perfectly*, and consider no setup parameters (in the subsequent sections, we remove these assumptions). Furthermore, we only address here the case where the security threshold is $t > n/3$. The case $t > n/4$ requires an additional generic step that we defer to the technical sections of the paper. We denote by $\Pi(\boldsymbol{v}; \boldsymbol{r})$ the output of an honest execution of Π on input $\boldsymbol{v} \in \{0,1\}^n$ and randomness $\boldsymbol{r}$ (each party P_i holds input v_i and randomness r_i). We let $\Pi(\boldsymbol{v})$ denote the resulting random variable determined by the parties' random coins, and we write $\Pi(\boldsymbol{v}) = b$ to denote the event that the parties output b in an honest execution of Π on input $\boldsymbol{v}$. All corrupt parties described below are locally consistent (see Section 1.1).

2.1 First-Round Halting

Assume the honest parties of Π halt at the end of the first round with probability $\gamma > 0$ when facing t corruptions (on every input). Our goal is to upperbound the value of γ. Our approach is inspired by the analogous lower-bound for deterministic protocols (cf., [25, 23]). Namely, we start with a configuration in which validity assures the common output is 0, and, while maintaining the same output, we gradually adjust it into a configuration in which validity assures the common output is 1, thus obtaining a contradiction. For randomized protocols, the challenge is to maintain the invariant of the output, even when the probability of halting is far from 1. We make the following observations:

$$\text{Almost pre-agreement:} \quad \operatorname{dist}(\boldsymbol{v}, b^n) \leq t \implies \Pi(\boldsymbol{v}) = b. \tag{1}$$

That is, in an honest execution of Π, if the parties almost start with preagreement, i.e., with at least $n - t$ of b's in the input vector, then the parties output b with probability 1. Equation 1 follows from *agreement* and *validity* by considering an adversary corrupting exactly those parties with input $v_i \neq b$, and otherwise not deviating from the protocol.

$$\text{Neighboring executions (N1):} \quad \text{dist}(\boldsymbol{v}_0, \boldsymbol{v}_1) \le t \implies \Pr_{\boldsymbol{r}}\left[\Pi(\boldsymbol{v}_0; \boldsymbol{r}) = \Pi(\boldsymbol{v}_1; \boldsymbol{r})\right] \ge \gamma. \quad (2)$$

That is, for two input vectors that are at most t-far (i.e., the resiliency threshold), the probability that the executions on these vectors yield the same output when using the same randomness is bounded below by the halting probability. To see why Equation 2 holds, consider the following adversary corrupting subset $\mathcal{C}$, for $\mathcal{C}$ being the set of indices where $\boldsymbol{v}$ and $\boldsymbol{v}'$ disagree. For an arbitrary partition $\{\overline{\mathcal{C}}_0, \overline{\mathcal{C}}_1\}$ of $\overline{\mathcal{C}}$, the adversary instructs $\mathcal{C}$ to send messages according to $\boldsymbol{v}_0$ to $\overline{\mathcal{C}}_0$ and according to $\boldsymbol{v}_1$ to $\overline{\mathcal{C}}_1$, respectively. With probability at least γ, all parties halt at the first round, and, by perfect agreement, all parties compute the same output.[7] Since parties in $\overline{\mathcal{C}}_b$ cannot distinguish this execution from a halting execution of $\Pi(\boldsymbol{v}_b; \boldsymbol{r})$, Equation 2 follows.

We deduce that if there are more than $n/3$ corrupt parties, then the halting probability is 0; this follows by combining the two observations above for $\boldsymbol{v}_0 = 0^{n-t}1^t$ and $\boldsymbol{v}_1 = 0^t 1^{n-t}$. Namely, by Equation 1, it holds that $\Pr_{\boldsymbol{r}}\left[\Pi(\boldsymbol{v}_0; \boldsymbol{r}) = \Pi(\boldsymbol{v}_1; \boldsymbol{r})\right] = 0$. Thus, by Equation 2, $\gamma = 0$.

2.2 Second-Round Halting – Arbitrary Protocols

We proceed to explain our bound for second-round halting of arbitrary protocols. Assume the honest parties of Π halt at the end of the second round with probability $\gamma > 0$ when facing t corruptions (on every input). Let $t = (1/3 + \varepsilon) \cdot n$, for an arbitrary small constant $\varepsilon > 0$. In spirit, the attack follows the footsteps of the single-round case described above; we show that neighboring executions compute the same output with good enough probability (related to the halting probability), and lower-bound the latter using the *almost pre-agreement* observation. There is, however, a crucial difference between the first-round and second-round cases; the honest parties can use the second round to detect whether (some) parties are sending inconsistent messages. Thus, the second round of the protocol can be used to "catch-and-discard" parties that are pretending to have different inputs to different parties, and so our previous attack breaks down. (In the one-round case, we exploit the fact that the honest parties cannot verify the consistency of the messages they received.) Still, we show that there is a suitable variant of the attack that violates the agreement of any "too-good" scheme.

At a very high level, the idea for proving the *neighboring* property is to *gradually* increase the set of honest parties towards which the adversary behaves according to $\boldsymbol{v}_1$ (for the remainder it behaves according to $\boldsymbol{v}_0$, which is a decreasing set of parties). While the honest parties might identify the attacking parties and discard their messages, they should still agree on the output and halt at the conclusion of the second round with high probability. We exploit this fact to show that at the two extremes (where the adversary is merely playing honestly according to $\boldsymbol{v}_0$ and $\boldsymbol{v}_1$, respectively), the honest parties behave essentially the same. Therefore, if at one extreme (for $\boldsymbol{v}_0$) the honest parties output b, it follows that they also output b at the other extreme (for $\boldsymbol{v}_1$), which proves the *neighboring* property for the second-round case.

[7] In the above, we have chosen to ignore a crucial subtlety. In an execution of the protocol, it may be the case that there is a suitable message (according to $\boldsymbol{v}_0$ or $\boldsymbol{v}_1$) to prevent halting, yet the adversary cannot determine which one to send. In further sections, we address this issue by taking a random partition of $\overline{\mathcal{C}}$ (rather than an arbitrary one). By doing so, we introduce an error-term of $1/2^{n-t}$ when we upper bound the halting probability γ.

We implement the above by augmenting the one-round attack as follows. In addition to corrupting a set of parties that feign different inputs to different parties, the adversary corrupts an extra set of parties that is inconsistent with regards to the messages it received from the first set of corrupted parties. To distinguish between the two sets of corrupted parties, the former (first) will be referred to as "pivot" parties (since they pivot their input) and will be denoted $\mathcal{P}$, and the latter will be referred to as "propagating" parties (since they carefully choose what message to propagate at the second round) and will be denoted $\mathcal{L}$. We emphasize that the propagating parties deviate from the protocol only at the second round and only with regards to the messages received by the pivot parties (not with regards to their input – as is the case for the pivot parties). In more detail, we partition $\overline{\mathcal{P}} = [n] \setminus \mathcal{P}$ into $\ell = \lceil 1/\varepsilon \rceil$ sets $\{\mathcal{L}_1, \ldots, \mathcal{L}_\ell\}$, and we show that, unless there exists i such that parties in $\mathcal{C} = \mathcal{P} \cup \mathcal{L}_i$ violate agreement (explained below), the following must hold for neighboring executions.

$$\text{Neighbouring executions (N2):} \quad \operatorname{dist}(\boldsymbol{v}_0, \boldsymbol{v}_1) \le n/3 \implies \tag{3}$$
$$\Pr[\Pi(\boldsymbol{v}_0) = b \text{ in two rounds}] \ge \Pr[\Pi(\boldsymbol{v}_1) = b \text{ in two rounds}] - 2(\ell+1)^2 \cdot (1-\gamma).$$

That is, for two input vectors that are at most $n/3$–far, the difference in probability that two distinct executions (for each input vector) yield the same output within two rounds is roughly upper-bounded by the quantity $(1-\gamma)/\varepsilon^2$ (i.e., non-halting probability divided by ε^2). To see that Equation 3 holds true, fix $\boldsymbol{v}_0, \boldsymbol{v}_1 \in \{0,1\}^n$ of hamming distance at most $n/3$, and let $\mathcal{P}$ be the set of indices where $\boldsymbol{v}_0$ and $\boldsymbol{v}_1$ differ. Consider the following $\ell + 1$ distinct variants of Π, denoted $\{\Pi_0, \ldots, \Pi_\ell\}$; in protocol Π_i, parties in $\mathcal{P}$ send messages to $\mathcal{L}_1, \ldots, \mathcal{L}_i$ according to the input prescribed by $\boldsymbol{v}_1$ and to $\mathcal{L}_{i+1}, \ldots, \mathcal{L}_\ell$ according to the input prescribed by $\boldsymbol{v}_0$, respectively. All other parties follow the instructions of Π for input $\boldsymbol{v}_0$. We write $\Pi_i = b$ to denote the event that the parties not in $\mathcal{P}$ output b. Notice that the endpoint executions Π_0 and Π_ℓ are identical to honest executions with input $\boldsymbol{v}_0$ and $\boldsymbol{v}_1$, respectively. Let Halt_i denote the event that the parties not in $\mathcal{P}$ halt at the second round in an execution of Π_i. We point out that $\Pr[\neg\mathsf{Halt}_i] \le (\ell+1) \cdot (1-\gamma)$, since otherwise the adversary corrupting $\mathcal{P}$ and running Π_i, for a random $i \in (\ell) := \{0, \ldots, \ell\}$, prevents halting with probability greater than $1 - \gamma$. Next, we inductively show that

$$\Pr[\Pi_i = b \wedge \mathsf{Halt}_i] \ge \Pr[\Pi_0 = b \wedge \mathsf{Halt}_0] - 2i \cdot (\ell+1) \cdot (1-\gamma), \tag{4}$$

for every $i \in (\ell)$, which yields the desired expression for $i = \ell$. In pursuit of contradiction, assume Equation 4 does not hold, and let i denote the smallest index for which it does not hold (observe that $i \neq 0$, by definition). Notice that

$$\begin{aligned}
&\Pr[(\Pi_{i-1} = b \wedge \mathsf{Halt}_{i-1}) \wedge (\Pi_i \neq b \wedge \mathsf{Halt}_i)] \\
&\quad \ge \Pr[\Pi_{i-1} = b \wedge \mathsf{Halt}_{i-1}] - \Pr[\Pi_i = b \vee \neg\mathsf{Halt}_i] \\
&\quad \ge \Pr[\Pi_{i-1} = b \wedge \mathsf{Halt}_{i-1}] - \Pr[\Pi_i = b \wedge \mathsf{Halt}_i] - \Pr[\neg\mathsf{Halt}_i] \\
&\quad > 2 \cdot (\ell+1) \cdot (1-\gamma) - \Pr[\neg\mathsf{Halt}_i] \\
&\quad \ge (\ell+1) \cdot (1-\gamma) > 0.
\end{aligned}$$

The second inequality follows from union bound and $A \vee \neg B \equiv (A \wedge B) \vee \neg B$, the third inequality is by induction hypothesis, and the last inequality by the bound $\Pr[\neg\mathsf{Halt}_i] \le (\ell+1) \cdot (1-\gamma)$.

It follows that an adversary corrupting $\mathcal{C} = \mathcal{P} \cup \mathcal{L}_i$ causes disagreement with non-zero probability by acting as follows: parties in $\mathcal{P}$ and $\mathcal{L}_i$ send messages according to Π_i and Π_{i-1} to $\overline{\mathcal{C}}_0$ and $\overline{\mathcal{C}}_1$, respectively, where $\{\overline{\mathcal{C}}_0, \overline{\mathcal{C}}_1\}$ is an arbitrary partition of $\overline{\mathcal{C}} = [n] \setminus \mathcal{P} \cup \mathcal{L}_i$. Since disagreement is ruled out by assumption, we deduce Equations 4 and 3. To conclude, we combine the *almost pre-agreement* property (Equation 1) with the *neighboring* property (Equation 3) with $\boldsymbol{v}_0 = 0^{n-t}1^t$, $\boldsymbol{v}_1 = 0^t 1^{n-t}$, and $b = 1$. Namely, $\Pr[\Pi(\boldsymbol{v}_0) = 1 \text{ in two rounds}] = 0$, by *almost pre-agreement* and $\Pr[\Pi(\boldsymbol{v}_1) = 1 \text{ in two rounds}] \geq \gamma$, by *almost pre-agreement* and *halting.* It follows that $0 \geq \gamma - 2(\ell+1)^2 \cdot (1-\gamma)$, by Equation 3, and thus $1 - \frac{1}{2(\ell+1)^2+1} \geq \gamma$, which yields the desired expression.

2.3 Second-Round Halting – Public-Randomness Protocols

In Section 2.2, we ruled out "very good" second-round halting for arbitrary protocols via an efficient locally consistent attack. Recall that if the halting probability is too good (probability almost one), then there is a somewhat simple attack that violates agreement and/or validity. In this subsection, we discuss ruling out *any* second-round halting, i.e., halting probability bounded away from zero, for public-randomness protocols.

We first explain why the attack – as is – does not rule out second-round halting. Suppose that at the first round the parties of Π send a deterministic function of their input, and at the second round they send the messages they received at the first round together with a uniform random bit. On input $\boldsymbol{v}$ and randomness $\boldsymbol{r}$, the parties are instructed *not* to halt at the second round (i.e., carry on beyond the second round until they reach agreement with validity) if a super-majority ($\geq n-t$) of the v_i's are in agreement and $\text{maj}(r_1, \ldots, r_n) \neq \text{maj}(v_1, \ldots, v_n)$, i.e., the majority of the random bits does not agree with the super-majority of the inputs. In all other cases, the parties are instructed to output $\text{maj}(r_1, \ldots, r_n)$. It is not hard to see that this protocol will halt with probability 1/2, even in the presence of the previous locally consistent adversary (regardless of the choice of propagating parties $\mathcal{L}_i$). More generally, if the randomness uniquely determines the output, the protocol designer can ensure that halting does not result in disagreement, by partitioning the randomness appropriately, and thus foiling the previous attack.[8]

To overcome the above apparent obstacle, we introduce another dimension to our locally consistent attack; we instruct an extra set of corrupted parties to abort at the second round without sending their second-round messages. By utilizing aborting parties, the adversary can potentially decouple the output/halting from the parties' randomness and thus either prevent halting or cause disagreement. In Section 2.3.1, we explain how to rule out second-round halting for a rather unrealistic class of public-randomness protocol. What makes the class of protocols unrealistic is that we assume security holds against unbounded locally consistent adversaries, and the protocol prescribes only a single bit of randomness per party per round. That being said, this case illustrates nicely our attack, and it also makes an interesting connection to Boolean functions analysis (namely, the KKL theorem [37]). For general public-randomness protocols, we only know how to analyze the aforementioned attack assuming Conjecture 5, as explained in Section 2.3.2.

[8] In Section 2.2, halting was close to 1 and thus the randomness was necessarily ambiguous regarding the output.

2.3.1 "Superb" Single-Coin Protocols

A BA protocol Π is *t-superb* if agreement and validity hold perfectly against an adaptive *unbounded* locally consistent adversary corrupting at most t parties, i.e., the probability that such an adversary violates agreement or validity is 0. A public-randomness protocol is *single-coin*, if, at any given round, each party samples a single unbiased bit.

▶ **Theorem 7** (Second-round halting, superb single-coin protocols)**.** *For every $\varepsilon > 0$ there exists $c > 0$ such that the following holds for large enough n. For $t = (1/3 + \varepsilon) \cdot n$, let Π be a t-superb, single-coin, n-party public-randomness Byzantine agreement protocol and let γ denote the probability that the protocol halts in the second round under a locally consistent attack. Then, $\gamma \leq n^{-c}$.*

We assume for simplicity that the parties do not sample any randomness at the first round, and write $\boldsymbol{r} \in \{0,1\}^n$ for the vector of bits sampled by the parties at the second round, i.e., r_i is a uniform random bit sampled by P_i.

As discussed above, our attack uses an additional set of corrupted parties of size $\sigma \cdot n$, dubbed the "aborting" parties and denoted $\mathcal{S}$, that abort indiscriminately at the second round (the value of σ is set to $\lfloor \varepsilon/4 \rfloor$ and $\ell = 2 \cdot \lceil 1/\varepsilon \rceil$ to accommodate for the new set of corrupted parties, i.e., $|\mathcal{L}_i| \leq n \cdot \varepsilon/2$). In more detail, analogously to the previous analysis, we consider $(\ell+1) \cdot \binom{n}{\sigma n}$ distinct variants of Π, denoted $\{\Pi_i^{\mathcal{S}}\}_{i,\mathcal{S}}$ and indexed by $i \in (\ell)$ and $\mathcal{S} \subseteq [n]$ of size σn, as follows. In protocol $\Pi_i^{\mathcal{S}}$, parties in $\mathcal{P}$ send messages to $\mathcal{L}_1, \ldots, \mathcal{L}_i$ according to the input prescribed by $\boldsymbol{v}_1$, and to $\mathcal{L}_{i+1}, \ldots, \mathcal{L}_\ell$ according to the input prescribed by $\boldsymbol{v}_0$ (recall that $\mathcal{P}$ is exactly those indices where $\boldsymbol{v}_0$ and $\boldsymbol{v}_1$ differ). Parties in $\mathcal{S}$ act according to $\mathcal{P}$ or $\mathcal{L}_j$, for the relevant j, except that they abort at the second round without sending their second-round messages. We write $\Pi_i^{\mathcal{S}}(\boldsymbol{r}) = b$ to denote the event that the parties not in $\mathcal{P} \cup \mathcal{S}$ output b, where the parties' second-round randomness is equal to $\boldsymbol{r}$. Let $\mathsf{Halt}_i^{\mathcal{S}}$ denote the event that all parties not in $\mathcal{P} \cup \mathcal{S}$ halt at the second round in an execution of $\Pi_i^{\mathcal{S}}$, and define $\mathcal{R}_i^{\mathcal{S}}(b) = \{\boldsymbol{r} \in \{0,1\}^n \text{ s.t. } \Pi_i^{\mathcal{S}}(\boldsymbol{r}) = b \wedge \mathsf{Halt}_i^{\mathcal{S}}\}$. The following holds:

Neighbouring executions (N2†): (5)

$$\forall \boldsymbol{v}_0, \boldsymbol{v}_1 \in \{0,1\}^n \text{ with } \operatorname{dist}(\boldsymbol{v}_0, \boldsymbol{v}_1) \leq n/3, \quad \forall b \in \{0,1\}, i \in [\ell] := \{1, \ldots, \ell\}:$$
$$\left(\forall \mathcal{S}\colon \Pr\left[\Pi_{i-1}^{\mathcal{S}} = b \wedge \mathsf{Halt}_{i-1}^{\mathcal{S}}\right] \geq \gamma/2\right) \implies \left(\forall \mathcal{S}\colon \Pr\left[\Pi_i^{\mathcal{S}} = b \wedge \mathsf{Halt}_i^{\mathcal{S}}\right] \geq \gamma/2\right).$$

In words, for both $b \in \{0,1\}$: if $\Pi_{i-1}^{\mathcal{S}} = b$ and halts in two rounds with large probability ($\geq \gamma/2$), for every $\mathcal{S}$, then $\Pi_i^{\mathcal{S}} = b$ and halts in two rounds with large probability, for every $\mathcal{S}$. Before proving Equation 5, we show how to use it to derive Theorem 7. We apply Equation 5 for $\boldsymbol{v}_0 = 0^{n-t}1^t$, $\boldsymbol{v}_1 = 0^t 1^{n-t}$, $b = 0$, and $i = \ell$, in combination with the properties of *validity* and *almost pre-agreement* (Equation 1). Namely, by these properties, a random execution of Π on input $\boldsymbol{v}_0$ where the parties in $\mathcal{S}$ abort at the second round yields output 0 with probability at least $\gamma/2$, for every $\mathcal{S} \in \binom{[n]}{\sigma n}$. Therefore, by Equation 5, we deduce that a random execution of Π on input $\boldsymbol{v}_1$ where the parties in $\mathcal{S}$ abort at the second round yields output 0 with probability at least $\gamma/2$, for every $\mathcal{S} \in \binom{[n]}{\sigma n}$. The latter violates either *validity* or *almost pre-agreement* – contradiction. To conclude the proof of Theorem 7, we prove Equation 5 by using the following corollary of the seminal KKL theorem [37] from Bourgain et al. [11]. (Recall that $\mathcal{R}|_{\overline{\mathcal{S}}}$ is the projection of $\mathcal{R}$ on the index-set $\overline{\mathcal{S}}$.)

▶ **Lemma 8.** *For every $\sigma, \delta \in (0,1)$, there exists $c > 0$ s.t. the following holds for large enough n. Let $\mathcal{R} \subseteq \{0,1\}^n$ be s.t. $|\mathcal{R}|_{\overline{\mathcal{S}}}| \leq (1-\delta) \cdot 2^{(1-\sigma)n}$, for every $\mathcal{S} \subseteq [n]$ of size σn. Then, $|\mathcal{R}| \leq n^{-c} \cdot 2^n$.*

Loosely speaking, Lemma 8 states that for a set $\mathcal{R} \subseteq \{0,1\}^n$, if the size of every projection on a constant fraction of indices is bounded away from one (in relative size), then the size of $\mathcal{R}$ is vanishingly small (again, in relative size).[9]

Going back to the proof, in pursuit of contradiction, let $i \geq 1$ denote the smallest index for which Equation 5 does not hold, and without loss of generality suppose $b = 0$, i.e., there exists $\mathcal{S}$ such that $|\mathcal{R}_i^{\mathcal{S}}(0)| < \gamma/2 \cdot 2^n$, and $|\mathcal{R}_{i-1}^{\mathcal{S}'}(0)| \geq \gamma/2 \cdot 2^n$, for every relevant $\mathcal{S}'$. We prove Equation 5 by proving Equations 6 and 7, which result in contradiction via Lemma 8.

$$\text{Halting:} \qquad |\mathcal{R}_i^{\mathcal{S}}(1)| \geq \gamma/2 \cdot 2^n \tag{6}$$

$$\text{Perfect agreement:} \qquad \forall \mathcal{S}': \quad |\mathcal{R}_i^{\mathcal{S}}(1)|_{\overline{\mathcal{S}'}}| \leq (1-\gamma/2) \cdot 2^{(1-\sigma)n} \tag{7}$$

Equation 6 follows by the *halting* property of $\Pi_i^{\mathcal{S}}$, since the execution halts if and only if $\boldsymbol{r} \in \mathcal{R}_i^{\mathcal{S}}(1) \cup \mathcal{R}_i^{\mathcal{S}}(0)$, and, by assumption, $|\mathcal{R}_i^{\mathcal{S}}(0)| < \gamma/2 \cdot 2^n$. To conclude, we prove Equation 7 by observing that for every $\mathcal{S}'$ and $b \in \{0,1\}$, and every $\boldsymbol{r}$ and $\boldsymbol{r}'$, if $\boldsymbol{r} \in \mathcal{R}_{i-1}^{\mathcal{S}'}(0)$ and $\boldsymbol{r}|_{\overline{\mathcal{S}'}} = \boldsymbol{r}'|_{\overline{\mathcal{S}'}}$, then $\boldsymbol{r}' \in \mathcal{R}_{i-1}^{\mathcal{S}'}(0)$ (by definition), i.e., membership to $\mathcal{R}_{i-1}^{\mathcal{S}'}(0)$ does not depend on the indices of $\mathcal{S}'$. It follows that $|\mathcal{R}_{i-1}^{\mathcal{S}}(0)|_{\overline{\mathcal{S}'}}| \geq \gamma/2 \cdot 2^{(1-\sigma)n}$, for every $\mathcal{S}'$, and therefore $|\mathcal{R}_i^{\mathcal{S}}(1)|_{\overline{\mathcal{S}'}}| \leq (1-\gamma/2) \cdot 2^{(1-\sigma)n}$, since the sets $\mathcal{R}_i^{\mathcal{S}}(1)|_{\overline{\mathcal{S}'}}$ and $\mathcal{R}_{i-1}^{\mathcal{S}}(0)|_{\overline{\mathcal{S}'}}$ are non-intersecting for every $\mathcal{S}'$. Otherwise, if $\mathcal{R}_i^{\mathcal{S}}(1)|_{\overline{\mathcal{S}'}} \cap \mathcal{R}_{i-1}^{\mathcal{S}}(0)|_{\overline{\mathcal{S}'}} \neq \emptyset$, then the following attack violates the superb quality of the protocol. Fix $\mathcal{S}'$ and $\boldsymbol{r}$ such that $\boldsymbol{r} \in \mathcal{R}_i^{\mathcal{S}}(1)$ and $\boldsymbol{r}|_{\overline{\mathcal{S}'}} \in \mathcal{R}_i^{\mathcal{S}}(1)|_{\overline{\mathcal{S}'}} \cap \mathcal{R}_{i-1}^{\mathcal{S}}(0)|_{\overline{\mathcal{S}'}}$, and consider the attacker controlling $\mathcal{P}$, $\mathcal{L}_i$, $\mathcal{S}$, and $\mathcal{S}'$ that sends messages according to $\Pi_i^{\mathcal{S}}$ and $\Pi_{i-1}^{\mathcal{S}'}$ to $\overline{\mathcal{C}}_0$ and $\overline{\mathcal{C}}_1$, respectively, where $\{\overline{\mathcal{C}}_0, \overline{\mathcal{C}}_1\}$ is an arbitrary partition of $\overline{\mathcal{C}} = [n] \setminus \mathcal{P} \cup \mathcal{L}_i \cup \mathcal{S} \cup \mathcal{S}'$. It is not hard to see the attacker violates agreement, whenever the randomness lands on $\boldsymbol{r}$.

▶ Remark 9. For superb, single-coin, public-randomness protocol, repeated application of Equation 2 and Lemma 8 rules out second-round halting for arbitrary (constant) fraction of corrupted parties (and not only $n/3$ fraction).

2.3.2 General (Public-Randomness) Protocols

The analysis above crucially relies on the superb properties of the protocol. While it can be generalized for protocols with near-perfect statistical security and constant-bit randomness, we only manage to analyze the most general case (i.e., protocols with non-perfect computational security and arbitrary-size randomness) assuming Conjecture 5. Very roughly (and somewhat inaccurately), when applying the above attack on general public-randomness protocols, the following happens for some $\delta > 0$ and both values of $b \in \{0,1\}$: for $(1-\delta)$-fraction of possible aborting subsets $\mathcal{S}$, the probability that the honest parties halt in two rounds and output the same value b, whether parties in $\mathcal{S}$ all abort or not, is bounded below by the halting probability. Assuming Conjecture 5, it follows that with probability δ over the randomness and $\mathcal{S}$, the honest parties under the attack output opposite values depending whether the parties in $\mathcal{S}$ abort or not. We conclude that the agreement of the protocol is at most δ.

References

1 Ittai Abraham, T.-H. Hubert Chan, Danny Dolev, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. Communication Complexity of Byzantine Agreement, Revisited. In *Proceedings of the 38th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 317–326, 2019.

[9] In the jargon of Boolean functions analysis, since every large set has a $o(n)$-size index-set of influence almost one, it follows that some projection on a constant fraction of indices is almost full.

2 Ittai Abraham, Srinivas Devadas, Danny Dolev, Kartik Nayak, and Ling Ren. Synchronous Byzantine Agreement with Expected $O(1)$ Rounds, Expected $o(n^2)$ Communication, and Optimal Resilience. In *Financial Cryptography and Data Security*, 2019.

3 Hagit Attiya and Keren Censor. Tight bounds for asynchronous randomized consensus. *Journal of the ACM*, 55(5):20:1–20:26, 2008.

4 Hagit Attiya and Keren Censor-Hillel. Lower Bounds for Randomized Consensus under a Weak Adversary. *SIAM Journal on Computing*, 39(8):3885–3904, 2010.

5 Ziv Bar-Joseph and Michael Ben-Or. A Tight Lower Bound for Randomized Synchronous Consensus. In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 193–199, 1998.

6 Michael Ben-Or. Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols (Extended Abstract). In *Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 27–30, 1983.

7 Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation (Extended Abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing (STOC)*, pages 1–10, 1988.

8 Michael Ben-Or and Nathan Linial. Collective Coin Flipping, Robust Voting Schemes and Minima of Banzhaf Values. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 408–416, 1985.

9 Michael Ben-Or, Elan Pavlov, and Vinod Vaikuntanathan. Byzantine agreement in the full-information model in o(log n) rounds. In *Proceedings of the 38th Annual ACM Symposium on Theory of Computing (STOC)*, pages 179–186, 2006.

10 Eli Ben-Sasson, Alessandro Chiesa, Matthew Green, Eran Tromer, and Madars Virza. Secure Sampling of Public Parameters for Succinct Zero Knowledge Proofs. In *IEEE Symposium on Security and Privacy*, pages 287–304, 2015.

11 Jean Bourgain, Jeff Kahn, and Gil Kalai. Influential coalitions for Boolean Functions. In *CoRR*, 2014. `arXiv:1409.3033`.

12 Sean Bowe, Ariel Gabizon, and Matthew D. Green. A Multi-party Protocol for Constructing the Public Parameters of the Pinocchio zk-SNARK. In *Financial Cryptography and Data Security FC*, pages 64–77, 2018.

13 Gabriel Bracha. An Asynchronou [(n-1)/3]-Resilient Consensus Protocol. In *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 154–162, 1984.

14 Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 173–186, 1999.

15 David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty Unconditionally Secure Protocols (Extended Abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing (STOC)*, pages 11–19, 1988.

16 Jing Chen and Silvio Micali. Algorand. In *CoRR*, 2016. `arXiv:1607.01341`.

17 Benny Chor and Brian A. Coan. A Simple and Efficient Randomized Byzantine Agreement Algorithm. In *Fourth Symposium on Reliability in Distributed Software and Database Systems, SRDS*, pages 98–106, 1984.

18 Benny Chor, Michael Merritt, and David B. Shmoys. Simple constant-time consensus protocols in realistic failure models. *Journal of the ACM*, 36(3):591–614, 1989.

19 Ran Cohen, Sandro Coretti, Juan Garay, and Vassilis Zikas. Round-Preserving Parallel Composition of Probabilistic-Termination Cryptographic Protocols. In *Proceedings of the 44th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 37:1–37:15, 2017.

20 Ran Cohen, Sandro Coretti, Juan A. Garay, and Vassilis Zikas. Probabilistic Termination and Composability of Cryptographic Protocols. In *Advances in Cryptology – CRYPTO 2016, part III*, pages 240–269, 2016.

21 Ran Cohen, Iftach Haitner, Nikolaos Makriyannis, Matan Orland, and Alex Samorodnitsky. On the Round Complexity of Randomized Byzantine Agreement. *CoRR*, abs/1907.11329, 2019.

22 Danny Dolev, Rüdiger Reischuk, and H. Raymond Strong. Early Stopping in Byzantine Agreement. *Journal of the ACM*, 37(4):720–741, 1990.

23 Danny Dolev and Raymond Strong. Authenticated Algorithms for Byzantine Agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.

24 Pesech Feldman and Silvio Micali. An Optimal Probabilistic Protocol for Synchronous Byzantine Agreement. *SIAM Journal on Computing*, 26(4):873–933, 1997.

25 Michael J. Fischer and Nancy A. Lynch. A Lower Bound for the Time to Assure Interactive Consistency. *Information Processing Letters*, 14(4):183–186, 1982.

26 Michael J. Fischer, Nancy A. Lynch, and Michael Merritt. Easy Impossibility Proofs for Distributed Consensus Problems. In *Proceedings of the 23th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 59–70, 1985.

27 Matthias Fitzi and Juan A. Garay. Efficient player-optimal protocols for strong and differential consensus. In *Proceedings of the 22th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 211–220, 2003.

28 Ehud Friedgut. Boolean Functions With Low Average Sensitivity Depend On Few Coordinates. *Combinatorica*, 18(1):27–35, 1998.

29 Juan A. Garay, Jonathan Katz, Chiu-Yuen Koo, and Rafail Ostrovsky. Round Complexity of Authenticated Broadcast with a Dishonest Majority. In *Proceedings of the 48th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 658–668, 2007.

30 Juan A. Garay and Yoram Moses. Fully polynomial Byzantine agreement in t+1 rounds. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing (STOC)*, pages 31–41, 1993.

31 Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, pages 51–68, 2017.

32 Oded Goldreich, Shafi Goldwasser, and Nathan Linial. Fault-Tolerant Computation in the Full Information Model. *SIAM Journal on Computing*, 27(2):506–544, 1998.

33 Oded Goldreich, Silvio Micali, and Avi Wigderson. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC)*, pages 218–229, 1987.

34 Shafi Goldwasser, Yael Tauman Kalai, and Sunoo Park. Adaptively Secure Coin-Flipping, Revisited. In *Proceedings of the 42th International Colloquium on Automata, Languages, and Programming (ICALP), part II*, pages 663–674, 2015.

35 Shafi Goldwasser, Elan Pavlov, and Vinod Vaikuntanathan. Fault-Tolerant Distributed Computing in Full-Information Networks. In *Proceedings of the 47th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 15–26, 2006.

36 Vassos Hadzilacos. Connectivity Requirements for Byzantine Agreement under Restricted Types of Failures. *Distributed Computing*, 2(2):95–103, 1987.

37 Jeff Kahn, Gil Kalai, and Nathan Linial. The Influence of Variables on Boolean Functions (Extended Abstract). In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 68–80, 1988.

38 Bruce M. Kapron, David Kempe, Valerie King, Jared Saia, and Vishal Sanwalani. Fast asynchronous Byzantine agreement and leader election with full information. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 1038–1047, 2008.

39 Anna R. Karlin and Andrew Chi-Chih Yao. Probabilistic lower bounds for Byzantine agreement and clock synchronization. Unpublished manuscript, 1986.

40 Jonathan Katz and Chiu-Yuen Koo. On Expected Constant-Round Protocols for Byzantine Agreement. In *Advances in Cryptology – CRYPTO 2006*, pages 445–462, 2006.

41 Valerie King and Jared Saia. Byzantine agreement in polynomial expected time: [extended abstract]. In *Proceedings of the 45th Annual ACM Symposium on Theory of Computing (STOC)*, pages 401–410, 2013.

42 John Kubiatowicz, David Bindel, Yan Chen, Steven E. Czerwinski, Patrick R. Eaton, Dennis Geels, Ramakrishna Gummadi, Sean C. Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Y. Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *ASPLOS-IX Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 190–201, 2000.

43 Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

44 Allison B. Lewko. The Contest between Simplicity and Efficiency in Asynchronous Byzantine Agreement. In *Proceedings of the 25th International Symposium on Distributed Computing (DISC)*, pages 348–362, 2011.

45 Allison B. Lewko and Mark Lewko. On the complexity of asynchronous agreement against powerful adversaries. In *Proceedings of the 32th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 280–289, 2013.

46 Yehuda Lindell, Anna Lysyanskaya, and Tal Rabin. On the Composition of Authenticated Byzantine Agreement. *Journal of the ACM*, 53(6):881–917, 2006.

47 Silvio Micali. Very Simple and Efficient Byzantine Agreement. In *Proceedings of the 8th Annual Innovations in Theoretical Computer Science (ITCS) conference*, pages 6:1–6:1, 2017.

48 Silvio Micali, Michael O. Rabin, and Salil P. Vadhan. Verifiable Random Functions. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 120–130, 1999.

49 Silvio Micali and Vinod Vaikuntanathan. Optimal and player-replaceable consensus with an honest majority. Unpublished manuscript, 2017.

50 Elchanan Mossel, Ryan O'Donnell, Oded Regev, Jeffrey E. Steif, and Benny Sudakov. Non-interactive correlation distillation, inhomogeneous Markov chains, and the reverse Bonami-Beckner inequality. *Israel Journal of Mathematics*, 154(1):299–336, 2006.

51 Elchanan Mossel, Krzysztof Oleszkiewicz, and Arnab Sen. On Reverse Hypercontractivity. *Geometric and Functional Analysis*, 23(3):1062–1097, 2013.

52 Gil Neiger and Sam Toueg. Automatically Increasing the Fault-Tolerance of Distributed Algorithms. *Journal of Algorithms*, 11(3):374–419, 1990.

53 Ryan O'Donnell. *Analysis of Boolean Functions*. Cambridge University Press, 2014.

54 Rafael Pass and Elaine Shi. Hybrid Consensus: Efficient Consensus in the Permissionless Model. In *Proceedings of the 31st International Symposium on Distributed Computing (DISC)*, pages 39:1–39:16, 2017.

55 Rafael Pass and Elaine Shi. Thunderella: Blockchains with Optimistic Instant Confirmation. In *Advances in Cryptology – EUROCRYPT 2018, part II*, pages 3–33, 2018.

56 Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching Agreement in the Presence of Faults. *Journal of the ACM*, 27(2):228–234, 1980.

57 Birgit Pfitzmann and Michael Waidner. Unconditional Byzantine Agreement for any Number of Faulty Processors. In *Proceedings of the 9th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 339–350, 1992.

58 Michael O. Rabin. Randomized Byzantine Generals. In *Proceedings of the 24th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 403–409, 1983.

59 Miklos Santha and Umesh V. Vazirani. Generating Quasi-Random Sequences from Slightly-Random Sources (Extended Abstract). In *Proceedings of the 25th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 434–440, 1984.

60 Andrew Chi-Chih Yao. Protocols for Secure Computations (Extended Abstract). In *Proceedings of the 23th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 160–164, 1982.

Trade-Offs in Distributed Interactive Proofs

Pierluigi Crescenzi
IRIF – CNRS and Université de Paris, France
piluc@irif.fr

Pierre Fraigniaud
IRIF – CNRS and Université de Paris, France
pierref@irif.fr

Ami Paz
IRIF – CNRS and Université de Paris, France
Faculty of Computer Science, University of Vienna, Austria
amipaz@irif.fr

Abstract

The study of interactive proofs in the context of distributed network computing is a novel topic, recently introduced by Kol, Oshman, and Saxena [PODC 2018]. In the spirit of sequential interactive proofs theory, we study the power of distributed interactive proofs. This is achieved via a series of results establishing trade-offs between various parameters impacting the power of interactive proofs, including the number of interactions, the certificate size, the communication complexity, and the form of randomness used. Our results also connect distributed interactive proofs with the established field of distributed verification. In general, our results contribute to providing structure to the landscape of distributed interactive proofs.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Theory of computation → Interactive proof systems; Theory of computation → Distributed algorithms

Keywords and phrases Distributed interactive proofs, Distributed verification

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.13

Related Version A full version of the paper is available at `https://arxiv.org/abs/1908.03363`.

Funding *Pierre Fraigniaud*: Additional support from the INRIA project GANG, and from the ANR project DESCARTES.
Ami Paz: Supported by the Fondation des Sciences Mathématiques de Paris.

Acknowledgements The authors are thankful to Gianlorenzo D'Angelo for fruitful discussions on the topic of this paper, and to Amir Yehudayoff for discussion on his work [27].

1 Introduction

This paper is concerned with distributed network computing, in which n processing nodes occupy the n vertices of a connected simple graph G, and communicate through the edges of G. In this context, *distributed decision* [25] refers to the task in which the nodes have to collectively decide whether the network G satisfies some given graph property, which may refer also to input labels given to the nodes (basic examples of such tasks are whether the network is acyclic or whether the network is properly colored). If the property is satisfied then all nodes must accept, otherwise at least one node must reject. Distributed decision finds immediate applications to distributed fault-tolerant computing, in which the nodes must check whether the current network configuration is in a legal state with respect to some Boolean predicate [22]. (If this is not the case, the rejecting node(s) may raise an alarm or launch a recovery procedure.)

© Pierluigi Crescenzi, Pierre Fraigniaud, and Ami Paz;
licensed under Creative Commons License CC-BY
33rd International Symposium on Distributed Computing (DISC 2019).
Editor: Jukka Suomela; Article No. 13; pp. 13:1–13:17

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

While some properties (e.g., whether a given coloring is proper) are locally decidable (LD) by exchanging information between neighbors only, other properties are not (e.g., whether the network is acyclic, or whether a given set of pointers forms a spanning tree of the network). As a remedy, the notion of *proof-labeling scheme* (PLS) was introduced [22], and variants were considered, including *non-deterministic local decision* (NLD) [14], and *locally checkable proofs* (LCP) [18]. All these settings assume the existence of a *prover* assigning *certificates* to the nodes, and a distributed *verifier* in charge of verifying that these certificates form a distributed proof that the network satisfies some given property. For instance, acyclicness can be certified by a prover picking one arbitrary node u, and assigning to each node v a certificate $c(v)$ equal its distance to u. The distributed verifier running at every node v checks that v has one neighbor w satisfying $c(w) = c(v) - 1$, and all its other neighbors w' satisfying $c(w') = c(v) + 1$. If the network contains a cycle, then these equalities will be violated in at least one node.

Note that the prover is not necessarily an abstract entity, as an algorithm constructing some distributed data structure (e.g., a spanning tree) may construct in parallel a proof that this data structure is correct. Interestingly, all (Turing decidable) graph properties can be certified by PLS and LCP with $O(n^2)$-bits certificates [22], and this is tight [18] – for instance, symmetry[1] (Sym) was shown to require $\Omega(n^2)$-bit certificates. However, not only such universal certification requires high space complexity at the nodes for storing large certificates, it also requires high communication complexity between neighbors for verifying the correctness of these certificates. Hence, the concern is minimizing the size of the certificates for specific graph properties, e.g., minimum-weight spanning trees (MST) [21].

Recently, the notion of *randomized* proof-labeling schemes (RPLS) was introduced [15]. RPLS assumes that the distributed verifier is randomized, and the global verdict provided by the nodes about the correctness of the network configuration should hold with probability at least $2/3$. Such randomized distributed verification schemes were proven to be very efficient in terms of communication complexity (with $O(\log n)$-bit messages exchanged between neighbors), but this often holds at the cost of actually *increasing* the size of the certificates provided by the prover.

Another recent direction for reducing the certificate size introduces a *local hierarchy* of complexity classes defined by *alternating quantifiers* (similarly to the polynomial hierarchy [28]) for local decision [11]. Interestingly, many properties requiring $\Omega(n^2)$-bit certificates with a locally checkable proof stand at the bottom levels of this hierarchy, with $O(\log n)$-bit certificates. This is for instance the case of non 3-colorability ($\overline{\mathsf{3Col}}$), which stands at the second level of the hierarchy, and Sym, which stands at the third level of the hierarchy. More generally, all monadic second order graph properties belong to this hierarchy with $O(\log n)$-bit certificates. However, it is not clear how to implement the protocols resulting from this hierarchy.

Even more recently, a very original and innovative approach was adopted [20, 24], bearing similarities with the local hierarchy but perhaps offering more algorithmic flavor. This approach considers *distributed interactive proofs.* Such proofs consist of a constant number of interactions between a centralized prover M (a.k.a. Merlin) and a randomized distributed verifier A (a.k.a. Arthur). For instance, a dAM protocol is a protocol with two interactions: Arthur queries Merlin by sending a random string, and Merlin replies to this query by sending a certificate. Similarly, a dMAM protocol involves three interactions: Merlin provides a

[1] G is symmetric if G has a non trivial automorphism, i.e., a one-to-one mapping from the set of nodes to itself preserving edges, and distinct from the identity map.

certificate to Arthur, then Arthur queries Merlin by sending a random string, and finally Merlin replies to Arthur's query by sending another certificate. This series of interactions is followed by a phase of distributed verification performed between every node and its neighbors, which may be either deterministic or randomized.

Although the interactive model seems weaker than the alternation of quantifiers in the local hierarchy, many properties requiring $\Omega(n^2)$-bits certificates with a locally checkable proof admit an Arthur-Merlin protocol with small certificates, and very few interactions. For instance, this is the case of Sym that admits a dMAM protocol with $O(\log n)$-bit certificates, and a dAM protocol with $O(n \log n)$-bit certificates [20] (on the other hand, any dAM protocol for Sym requires $\Omega(\log \log n)$-bit certificates [20]). It is also known that non symmetry ($\overline{\mathsf{Sym}}$) can be decided by a dAMAM protocol with $O(\log n)$-bit certificates [24]. These results raise several interesting questions, such as:

1. Are there ways to establish trade-offs between space complexity (i.e., the size of the certificates) and communication complexity (i.e., the size of the messages exchanged between nodes)? The dMA protocols [20] as well as the RPLS protocols [15] enable to gain a lot in terms of message complexity, but at the cost of still high space complexity. Would it be possible to compromise between these two complexities? In particular, would it be possible to reduce the certificate size at the cost of increasing the communication complexity?
2. The theory of distributed decision has somehow restricted itself to distributed randomness, in the sense that each node has only access to a private source of random coins. These coins are public to the prover, but remain private to the other nodes. Shared randomness is known to be stronger than private randomness for communication complexity, as witnessed by, e.g., deciding equality [23]. How much shared randomness could help in the context of distributed decision?
3. Last but not least, are there general reduction theorems between Arthur-Merlin classes for trading the number of interactions with the certificate size? In the centralized setting, it is known that $\mathsf{AM}[k] = \mathsf{AM}[2]$ for any $k \geq 2$, but it is not known whether a similar claim holds in the distributed setting [20]. Also, in the centralized setting the Sipser–Lautemann theorem tells us that $\mathsf{MA} \subseteq \Sigma_2 \cap \Pi_2$ and $\mathsf{MA} \subseteq \mathsf{AM} \subseteq \Pi_2$, but it is not known whether the distributed Arthur-Merlin classes stand so low in the local alternating hierarchy too.

Our Results

In this paper, we study the power of distributed interactive proofs. This is achieved via a series of results establishing trade-offs between various parameters impacting the power of interactive proofs, including the number of interactions, the certificate size, the communication complexity, and the form of randomness used. Our results also connect distributed interactive proofs with the established field of distributed verification. We address the above three questions as follows. For the first question, we show how to apply techniques developed in the framework of multi-party communication complexity to get trade-offs between space and communication for the classical triangle detection problem. For the second question, we show that shared randomness helps significantly, enabling to exponentially reduce the communication complexity while preserving the space complexity for important problems such as spanning tree, and a vast class of optimization problems, including, for example, maximum independent set and minimum dominating set. For the third question, we give a general technique for reducing the number of interactions at the cost of increasing the certificate and message size.

More specifically, for the first question, we explore the trade-off of space vs. communication, and establish that for every α there exists a Merlin-Arthur protocol for triangle-freeness, which uses $O(\log n)$ bits of shared randomness, $\widetilde{O}(n/\alpha)$-bit certificates and $\widetilde{O}(\alpha)$-bit messages between nodes (Theorem 4). To our knowledge, this is the first example of a decision task for which one can trade communication for space. In addition, the proof reveals an interesting connection between dMA and communication complexity with a referee. Note that, for $\alpha = \sqrt{n}$, we obtain a distributed Merlin-Arthur protocol for triangle-freeness with message and space complexities $\widetilde{O}(\sqrt{n})$ bits. In contrast, any proof-labeling scheme for triangle-freeness must have certificate size at least $n/e^{O(\sqrt{\log n})}$ bits (Proposition 5). A similar tradeoff can be obtained when using distributed randomness, though with higher space complexity.

Regarding the second question, we explore the significance of having access to shared randomness. We show that, for any minimization problem π in graphs whose admissibility can be decided locally, there exists a Merlin-Arthur protocol for certifying the existence of a solution whose cost is at most k, using $O(\log n)$ bits of shared randomness, with $O(\log n)$-bit certificates and $O(\log \log n)$-bit messages between nodes (see Theorem 6). The same result holds for maximization problems whose admissibility can be decided locally. Note that this class of problems includes, for example, maximum independent set, minimum dominating set, and minimum vertex cover (potentially weighted). This exponentially improves the communication complexity of locally checkable proofs for such problems. The same communication complexity could be obtained using randomized proof-labeling schemes, but at the cost of increasing the certificate size to up to $O(n \log n)$ bits. As another interesting result in the context of exploring the significance of having access to shared randomness, we show that even shared randomness remains limited both in terms of the certificate size and of the amount of communication. We show that every Arthur-Merlin protocol for Sym, and every Arthur-Merlin protocol for $\overline{\text{Sym}}$ must have both certificate size and message size $\Omega(\log \log n)$ bits, even with shared randomness (see Theorem 10). Interestingly, for the class of graphs used in the proof of this latter result, there is a Merlin-Arthur protocol with certificates and messages of constant length. This shows that the inclusion $\mathsf{MA} \subseteq \mathsf{AM}$ which holds in the centralized setting does not hold in the distributed setting.

Finally, we consider general reductions within the Arthur-Merlin hierarchy, and compare the power of this hierarchy to the power of proof-labeling schemes with certificates of linear size. We show that, for every σ and γ, any graph property verifiable with an Arthur-Merlin protocol with 3 or 4 interactions (dMAM or dAMAM) using σ-bit certificates and γ-bit messages can also be verified by an Arthur-Merlin (dAM) protocol using $O(n\sigma^2)$-bit certificates and $O(n\gamma\sigma)$-bit messages (see Theorem 11 and Corollary 12). Although the linear blowup in terms of both certificate size and message complexity may seem huge at a first glance, it fits (up to logarithmic factors) with the different results obtained previously [20, 24] regarding Sym and graph non-isomorphism ($\overline{\text{Iso}}$). Finally, we compare the power of Arthur-Merlin protocols with an arbitrarily large number of interactions with the power of proof-labeling schemes. We show that there exists a graph property admitting a proof-labeling scheme with certificates and messages on $O(n)$ bits, that cannot be solved by an Arthur-Merlin protocol with $o(n)$-bit certificates, for any fixed number $k \geq 0$ of interactions between Arthur and Merlin, even using shared randomness, and even with messages of unbounded size (see Theorem 14). This latter result demonstrates that, in general, one cannot trade the number of interactions between Merlin and Arthur for reducing the certificate size, at least for certificates of linear size.

Most of our results are stated by assuming that nodes have access to shared randomness. However, as all our protocols are local, all our 1-round protocols can be simulated by 2-round protocols using distributed randomness. In general, our results contribute to providing structure to the landscape of distributed interactive proofs.

Related Work

Local decision (LD) and the central notion of locally-checkable labellings were introduced and thoroughly studied in the 90s [25]. Local verification was introduced fifteen years later [22], through the original notion of proof-labeling schemes (PLS). Proof-labeling schemes find important applications to self-stabilization, but are subject to some restrictions (only certificates are exchanged between neighbors). These restrictions were lifted by considering the general notion of locally checkable proofs (LCP) [18]. By definition, we have $\mathsf{LCP} = \Sigma_1\mathsf{LD}$ (the same way $\mathsf{NP} = \Sigma_1\mathsf{P}$). A third notion of distributed verification was introduced, by considering the class NLD [14]. NLD differs from $\Sigma_1\mathsf{LD}$ in the fact that, in NLD, the certificates cannot depend on the identities given to the nodes.

Randomized versions of local decision and local verification have been considered in the literature [15, 13, 14, 20]. A Merlin-Arthur (dMA) protocol is actually a randomized version of locally checkable proof ($\Sigma_1\mathsf{LD}$) that was previously studied [15]: Merlin provides each node with a certificate, and Arthur performs a randomized verification algorithm at each node. The benefit of using dMA over $\Sigma_1\mathsf{LD}$ can be exponential in terms of communication complexity (i.e., of the size of the messages exchanged between neighbors), at the cost of a linear increase in space complexity (i.e., of the size of the certificates provided by Merlin) [15].

Very closely related to the line of work about interactive distributed proofs is the local hierarchy $\mathsf{LH} = \bigcup_{k\geq 0}\left(\Sigma_k\mathsf{LD} \cup \Pi_k\mathsf{LD}\right)$ with certificates and messages of logarithmic size [11], extending the known logLCP class [18]. In particular, it is proved that $\mathsf{Sym} \in \Sigma_3\mathsf{LD}$ [11]. Also, it is easy to show that $\overline{\mathsf{3Col}} \in \Pi_2\mathsf{LD}$. In contrast, placing Sym or $\overline{\mathsf{3Col}}$ in $\Sigma_1\mathsf{LD}$ requires $\Omega(n^2)$-bit certificates [18]. The same hierarchy was later considered, but under the constraint that the certificates must not depend on the identifier assignment to the nodes. With $O(n^2)$-bit certificates, this hierarchy collapses at the second level $\Pi_2\mathsf{LD}$ [5]. (This is in contrast with the hierarchy in which certificates can depend on the node identifiers, which collapses at the first level $\Sigma_1\mathsf{LD}$ with $O(n^2)$-bit certificates.) Nevertheless, apart from the bottom levels, the hierarchies based on alternating quantifiers with $O(\log n)$-bit certificates are essentially the same [5]. See the recent survey [10] for more results on distributed decision.

This work is inspired by very recent achievements in the field of distributed interactive protocols [20, 24]. In addition to the aforementioned results regarding Sym and $\overline{\mathsf{Sym}}$, two versions of $\overline{\mathsf{Iso}}$ (Given two sub-graphs G_1 and G_2 of G, $G_1 \not\simeq G_2$, that is, are G_1 and G_2 non-isomorphic?) were considered. For the easiest version, in which the input of each node v is formed by the two sets of neighbors of v in the two graphs G_1 and G_2, a dAMAM protocol with $O(\log n)$-bit certificates for deciding $G_1 \not\simeq G_2$ was designed [24]. For the more complicated version, in which $G_1 = G$ (that is, G_1 is the communication network) and the input of each node v is the set of neighbors of v in G_2, a dAMAM protocol with $O(n \log n)$-bit certificates for deciding $G_1 \not\simeq G_2$ was proposed [20], and an Arthur-Merlin protocol with a constant number of interactions and $O(\log n)$-bit certificates, for deciding $G_1 \not\simeq G_2$, was successively designed [24]. Interestingly, this latter result is obtained via a general connection between efficient centralized computation (under various models) and the ability to design Arthur-Merlin protocols with a constant number of interactions between the prover and the verifier, using logarithmic-size certificates.

We use a variety of techniques and results from the theory of *communication complexity* [23]. Specifically, we use an Arthur-Merlin style protocol for two-party disjointness [1], in a recent variant [2] that allows a trade-off between communication complexity and certificate size. We also use recent lower bounds for the equality and non-equality problems in the same setting [17]. Finally, we use multi-party communication protocol with a referee for the SumZero problem with bounded inputs [26], and with unbounded inputs [19].

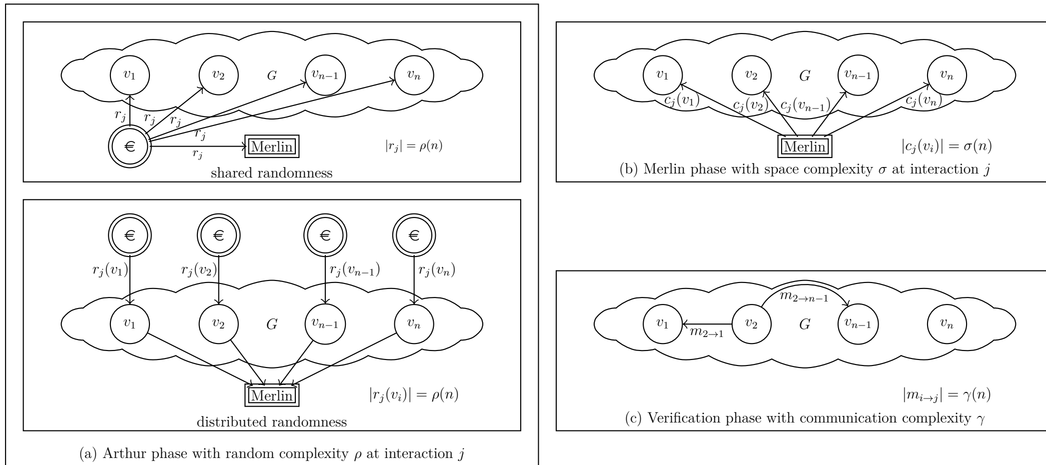


Figure 1 The three different phases of a distributed Arthur-Merlin protocol (the Arthur phase can make use of either shared or distributed randomness).

2 Model and Definitions

A *network configuration* is a triple (G, id, x) where $G = (V, E)$ is a connected simple graph, $\mathrm{id} : V \to \{1, \dots, n^c\}$ for some constant $c \geq 1$ is the *identity* one-to-one assignment to the nodes, and $x : V \to \{0, 1\}^*$ is the *input label* assignment (i.e., the state of the node). A *distributed language* is a collection $\mathcal{L}$ of network configurations. Note that it may be the case that, for some language $\mathcal{L}$, $(G, \mathrm{id}, x) \in \mathcal{L}$ while $(G, \mathrm{id}', x) \notin \mathcal{L}$ for two different identity assignments id and id$'$. This typically occurs for languages where the label of a node refers to the identities of its neighbors, e.g., for encoding spanning trees. (Throughout the paper, we assume that all considered distributed languages are Turing-decidable). *Distributed decision* for $\mathcal{L}$ is the following task: given any network configuration (G, id, x), the nodes of G must collectively decide whether $(G, \mathrm{id}, x) \in \mathcal{L}$. If this is the case, then *all* nodes must accept, otherwise *at least one node* must reject (with certain probabilities, depending on the model).

We consider *interactive protocols* for distributed decision [20]. A distributed interactive protocol $\mathcal{P}$ consists of a constant series of interactions between a *prover* called *Merlin*, and a *verifier* called *Arthur* (see Fig. 1 for a visual representation of such a protocol). The prover Merlin is centralized, and has unlimited computing power. It is aware of the whole network configuration (G, id, x) under consideration, but it cannot be trusted. The verifier Arthur is distributed, and has bounded knowledge, that is, at each node v, Arthur is initially aware solely of $(\mathrm{id}(v), x(v))$, i.e., of its identity and its input label.

In a distributed *Arthur-Merlin* interactive protocol performed on $\mathcal{I} = (G, \mathrm{id}, x)$, whenever Arthur is the one that starts interacting, it picks a random string $r_1(v)$ at each node v of G (this random string might be private to each node, or the nodes may have access to shared randomness). Given the collection r_1 of random strings selected by the nodes, Merlin provides every node v with a *certificate* $c_1(v) = \mathsf{p}(v, \mathcal{I}, r_1)$, where $\mathsf{p} : \{0, 1\}^* \to \{0, 1\}^*$. At this point, Arthur picks another random string $r_2(v)$ at each node v. Then Merlin replies to each node v by sending a second certificate $c_2(v) = \mathsf{p}(v, \mathcal{I}, r_1, r_2)$, and so on. Whenever Merlin is the one that starts interacting, the process starts with Merlin constructing a binary string $c_0(v) = \mathsf{p}(v, \mathcal{I})$ that it sends to every node v. These interactions proceed for a constant number

$k \geq 0$ of times, and Merlin interacts last, by sending $c_{\lfloor \frac{k}{2} \rfloor}(v)$ to every node v. A sequence of interactions can then be summarized by a *transcript* $\pi(\mathcal{I}, \mathsf{p}, r) = \Big(\pi_v(\mathcal{I}, \mathsf{p}, r)\Big)_{v \in V}$, where $r = \Big(r_1, \ldots, r_{\lfloor \frac{k}{2} \rfloor}\Big)$ with $r_i = (r_i(v))_{v \in V}$ for $i = 1, \ldots, \lfloor \frac{k}{2} \rfloor$, and

$$\pi_v(\mathcal{I}, \mathsf{p}, r) = \Big(c_0(v), r_1(v), c_1(v), \ldots, r_{\lfloor \frac{k}{2} \rfloor}(v), c_{\lfloor \frac{k}{2} \rfloor}(v)\Big)$$

with $c_0(v) = \varnothing$ if Arthur starts the interactions. In other words, an Arthur-Merlin protocol with k interactions results in a transcript with $c_0(v) = \varnothing$ if k is even, and with $c_0(v)$ equal to the first certificate provided by Merlin otherwise. The Arthur-Merlin protocol completes by performing a *deterministic* distributed verification algorithm v executed at each node. Specifically, Algorithm v proceeds as follows at every node v:

1. A message $M_{v,u}$ destined for every neighboring node u of v is forged, and sent to u. This message may depend on the identity $\mathrm{id}(v)$, the input $x(v)$, all random strings generated by Arthur at v, and all certificates received by v from Merlin.
2. Based on all the knowledge accumulated by v (i.e., its identity, its input label, the generated random strings, the certificates received from Merlin, and all the messages received from its neighbors), Algorithm v accepts or rejects at node v.

A distributed Arthur-Merlin protocol $\mathcal{P}$ thus consists of two consecutive stages: (1) interactions between the nodes and the prover p (Arthur-Merlin phases), and (2) communication among neighboring nodes (algorithm v). Note that, for the sake of simplifying the presentation, and unifying the comparison with previous work, we restrict ourselves to verification algorithms v that perform in a single round. Performing more than one round enables to improve the complexity of verification protocols in some cases [12]. However, this does not conceptually change the nature of the protocol. For zero interactions (i.e., $k = 0$), a distributed Arthur-Merlin protocol simply consists in performing a (deterministic) decision algorithm at each node [22]. For one interaction (i.e., $k = 1$), a distributed Arthur-Merlin verification protocol is a (1-round) locally-checkable proof algorithm [18].

▶ **Definition 1.** *The class* $\mathsf{dAM}[k](\sigma, \gamma)$ *is the class of languages* $\mathcal{L}$ *for which there exists a distributed Arthur-Merlin verification protocol with at most* $k \geq 0$ *interactions between Arthur and Merlin, where Merlin provides certificates of at most* $\sigma \geq 0$ *bits to the nodes, and the verification algorithm* v *exchanges messages of at most* $\gamma \geq 0$ *bits between nodes, such that, for every configuration* $\mathcal{I} = (G, \mathrm{id}, x)$,

$$\begin{cases} (G, \mathrm{id}, x) \in \mathcal{L} & \Rightarrow & \exists \mathsf{p} : \Pr_r[\mathsf{v}(\pi(\mathcal{I}, \mathsf{p}, r)) \text{ accepts at all nodes}] \geq 2/3; \\ (G, \mathrm{id}, x) \notin \mathcal{L} & \Rightarrow & \forall \mathsf{p} : \Pr_r[\mathsf{v}(\pi(\mathcal{I}, \mathsf{p}, r)) \text{ rejects in at least one node}] \geq 2/3. \end{cases}$$

The definition of distributed *Merlin-Arthur* interactive protocols, and of $\mathsf{dMA}[k](\sigma, \gamma)$ is similar, apart from the fact that, as opposed to Arthur-Merlin protocols in which Merlin always interacts last, Arthur has one more “interaction” during which it picks a random bit-string $r'(v)$ at every node v, which is used to perform a *randomized* verification algorithm v. Therefore, for $k \geq 1$, a Merlin-Arthur protocol with k interactions can be defined as an Arthur-Merlin protocol with $k - 1$ interactions, but where the verification algorithm v is randomized. For zero interactions, a distributed Merlin-Arthur protocol simply consists in performing a (deterministic) decision algorithm at each node [22]. For one interaction, a distributed Merlin-Arthur protocol is a (1-round) randomized decision algorithm as studied previously [13]. For two interactions, a distributed Merlin-Arthur protocol is a (1-round) randomized locally-checkable proof algorithm, also as studied previously [15].

In the following, we may avoid mentioning the parameters σ and γ when they are clear from the context, or when they are respectively identical in the two terms of an equality. For small values of $k \geq 2$, $\mathsf{dAM}[k]$ and $\mathsf{dMA}[k]$ are rewritten as an alternating sequence of As and Ms. For instance, $\mathsf{dAM}[2] = \mathsf{dAM}$, $\mathsf{dMA}[2] = \mathsf{dMA}$, $\mathsf{dAM}[3] = \mathsf{dMAM}$, $\mathsf{dMA}[3] = \mathsf{dAMA}$, and $\mathsf{dAM}[4] = \mathsf{dAMAM}$, and so on. For $k \leq 1$, it follows from the definition that $\mathsf{dAM}[0] = \mathsf{dMA}[0] = \mathsf{LD}$. We also have $\mathsf{dAM}[1] = \Sigma_1\mathsf{LD}$ and $\mathsf{dMA}[1] = \mathsf{BPLD}(2/3, 2/3)$ where the class $\mathsf{BPLD}(p, q)$ is the distributed version of BPP [16], with p being the acceptance probability of the interactive protocol on legal instances, and q being the rejection probability of the protocol on illegal instances [13]. As a last example, dMA is the class of languages that can be decided by a randomized locally checkable proof, as studied previously [15].

As opposed to the sequential setting in which it is known that $\mathsf{AM}[k] = \mathsf{AM}[2]$ for all $k \geq 2$, it is not known whether such "collapse" occurs in the distributed setting. Therefore, we define the Arthur-Merlin *hierarchy* as $\mathsf{dAMH}(\sigma, \gamma) = \bigcup_{k \geq 0} \mathsf{dAM}[k](\sigma, \gamma)$. That is, $\mathcal{L} \in \mathsf{dAMH}(\sigma, \gamma)$ if and only if there exists $k \geq 0$ such that $\mathcal{L} \in \mathsf{dAM}[k](\sigma, \gamma)$.

Boosting the Success Probability

In classical, sequential randomized algorithm, the success probability constant $2/3$ can be easily increased using repetitions. On the other hand, it was shown that this boosting technique is not applicable for randomized distributed decision algorithms in general [13], making the choice of constant significant when considering such settings. The inability of boosting in the distributed setting is due to the fact that, when repeating the algorithm on a "no" instance for several times, different nodes may reject in different repetitions, causing each node to sees very few rejections and decide on acceptance. Somewhat surprisingly, we can show that in the case of distributed Arthur-Merlin protocols (i.e., $\mathsf{dAM}[k]$ classes), parallel repetition is possible, and at a relatively low blowup in communication and certificates. This allows us to boost the success probability, as follows.

▶ **Proposition 2.** *Let $1 > p' > p > 1/2$. If there exists an Arthur-Merlin verification protocol $\mathcal{P}$ with $k \geq 2$ interactions that enables to verify a distributed language $\mathcal{L}$ with σ-bit certificates, γ-bit messages, and success probability p, then there exists an Arthur-Merlin verification protocol $\mathcal{P}'$ with k interactions that enables to verify $\mathcal{L}$ with $\sigma + O(\log n)$-bit certificates, messages on $\gamma + O(\log n)$ bits, and success probability p'.*

Proof. Moving from success probability p to success probability $p' > p$ is achieved in a standard way, by merely repeating $\mathcal{P}$ a constant number of times (depending on p and p'), and adopting the majority of the outcomes. However, this cannot be done in a straightforward manner because, for a configuration $(G, x, \mathrm{id}) \notin \mathcal{L}$, it may be the case that the (at least one) node rejecting (G, x, id) is different at each repetition. Therefore, during the last interaction with the prover, Merlin provides every node with a local encoding of a spanning tree T enabling to count the number of executions of $\mathcal{P}$ resulting in at least one node rejecting. It is known that certificates of $O(\log n)$ bits suffice for certifying such a tree [22]. The root of the tree T accepts or rejects depending on whether the majority of executions of $\mathcal{P}$ accepted or rejected, respectively. ◀

3 Space vs. Communication

In this section, we study the trade-off between space and communication complexity for Merlin-Arthur interactive protocols. Specifically, we consider the classical triangle-freeness problem, and establish a trade-off between space and communication for this problem. Recall

that a graph $G = (V, E)$ is triangle-free if, for every three nodes u, v, w in V, either $\{u, v\} \notin E$, $\{u, w\} \notin E$, or $\{v, w\} \notin E$. We denote by Δ_{free} the corresponding distributed language. There is a recent deterministic distributed algorithm for triangle-freeness running in $\widetilde{O}(\sqrt{n})$ rounds in the CONGEST model [7]. A general scheme for designing dMA protocol has also been proposed [15]. This scheme enables to reduce communication complexity to $O(\log n)$, at the cost of increasing the space complexity to $O(n^2)$. When more interactions are allowed, say a constant number k, a recent reduction [24] – from centralized small-space algorithms to our setting – implies a protocol with $O(\log n)$-bit certificates and $O(\log n)$-bit messages for the triangle-freeness problem, i.e., $\Delta_{\mathsf{free}} \in \mathsf{dMA}[k](\log n,\ \log n)$ for some constant $k > 1$.

In order to prove the trade-off between space and communication complexities for triangle-freeness, we first state the following result, which will be used at several places in the paper.

▶ **Lemma 3.** *For any network of maximum degree d, there exists a proof-labeling scheme (and thus a $\Sigma_1\mathsf{LD}$ protocol) with $O(\log n)$-bit certificates providing each node v with the certified value n of the number of nodes, and a color $c(v) \in \{1, \dots, \min\{d^2 + 1, n\}\}$ such that c forms a certified proper distance-2 coloring of the network.*

Proof idea. The certification of the number of nodes can be done by using a rooted spanning tree and by counting nodes in the sub-trees. The certification of a proper distance-2 coloring can be done by assigning colors to nodes, with every node checking that all its neighbors have different colors, all different from its own color. ◀

Since triangle-freeness is a local property, nodes do not need to be represented by identifiers that are different throughout the entire network. Instead, identifiers resulting from a proper distance-2 coloring suffice. Therefore, using Lemma 3, we can assume that nodes are provided with identifiers in $\{1, \dots, n\}$ such that $\mathrm{id}(u) \neq \mathrm{id}(v)$ whenever the distance between u and v is at most 2.

▶ **Theorem 4.** *For every $\alpha = O(n)$, there exists a Merlin-Arthur protocol for triangle-freeness, using $O(\log n)$ bits of shared randomness, with $O(\frac{n}{\alpha} \log n)$-bit certificates and $O(\alpha \log n)$-bit messages between nodes. In short $\Delta_{\mathsf{free}} \in \mathsf{dMA}(\frac{n}{\alpha} \log n,\ \alpha \log n)$.*

Proof. We identify the space $\{1, \dots, n\}$ of IDs with $[n/\alpha] \times [\alpha]$, for some $\alpha = O(n)$ of choice. Each node u thus has a set S_u of pairs of the form (i, t) representing its neighbors. Let q be a prime such that $cn\alpha < q \leq 2cn\alpha$, for a large enough constant $c > 1$, and let $\mathbb{F}_q$ be the field of q elements. Each node u represents S_u as α functions $\psi_{S_u,t} : [n/\alpha] \to \{0, 1\}$, where $\psi_{S_u,t}(i) = 1 \iff (i, t) \in S_u$. Node u then extends these functions to polynomials $\Psi_{S_u,t} : \mathbb{F}_q \to \mathbb{F}_q$ of degree at most $n/\alpha - 1$ that agree with $\psi_{S_u,t}$ on $[n/\alpha]$. To make sure that an edge $\{u, v\}$ is not a part of a triangle, the nodes u and v need to verify that $S_u \cap S_v = \emptyset$, which is equivalent to $\Psi_{S_u,t}(i) \cdot \Psi_{S_v,t}(i) = 0$ for all $i \in [n/\alpha]$ and $t \in [\alpha]$. Node u then defines its *neighbors polynomials* $\Psi_{uv,t} = \Psi_{S_u,t} \cdot \Psi_{S_v,t}$ for every $v \in S_u$, and every $t \in [\alpha]$. Let $\Psi_u = \sum_{t \in [\alpha]} \sum_{v \in S_u} \Psi_{uv,t}$. The degree of each polynomial $\Psi_{uv,t}$ is at most $2(n/\alpha - 1)$, and thus this is also the case for the degree of Ψ_u. Node u is not part of a triangle if and only if $\Psi_{uv,t}(i) = 0$ for every $t \in [\alpha]$, $i \in [n/\alpha]$ and $v \in S_u$. Since $q > n\alpha$, it follows that u is not part of a triangle if and only if $\Psi_u(i) = 0$ for every $i \in [n/\alpha]$. (For each i, we have a sum of $n\alpha$ values, each in $\{0, 1\}$.)

Merlin assigns to node u the certificate Φ_u, which is supposed to be equal to Ψ_u. Since this is a polynomial of degree at most $2(\frac{n}{\alpha} - 1)$, the same number of coefficients are sufficient for representing Φ_u. Therefore, the certificates are of $O(\frac{n}{\alpha} \log q)$ bits, which are actually $O(\frac{n}{\alpha} \log n)$ bits, as $q \leq 2cn\alpha = O(n^2)$.

Each node u first verifies that $\Phi_u(i) = 0$ for every $i \in [n/\alpha]$. Then, it checks that indeed $\Phi_u = \Psi_u$, as follows. The protocol uses the shared randomness to choose a field element $i_0 \in \mathbb{F}_q$ known to all nodes. Each node v broadcasts $\{\Psi_{S_v,t}(i_0) : t \in [\alpha]\}$ to each of its neighbors, using $O(\alpha \log q) \leq O(\alpha \log n)$ bits of communication. Node u then computes

$$\Psi_u(i_0) = \sum_{t\in[\alpha]} \sum_{v\in S_u} \Psi_{uv,t}(i_0) = \sum_{t\in[\alpha]} \sum_{v\in S_u} \Psi_{S_u,t}(i_0) \cdot \Psi_{S_v,t}(i_0)$$

and accepts only if $\Phi_u(i_0) = \Psi_u(i_0)$. The probability that two non-equal polynomials on $\mathbb{F}_q$ of degree at most $2(\frac{n}{\alpha} - 1)$ are equal at a random point i is at most $2(\frac{n}{\alpha} - 1)/q$. Since $q > cn\alpha$, the probability of error can be made arbitrarily small by choosing c large enough. ◀

▶ Remark. Similar trade-offs can still be obtained even if nodes have only access to distributed randomness. For instance, in two rounds, with the same notations as in the proof of Theorem 4, we can have each node u choose its own random $i_u \in \mathbb{F}_q$, and send it to all its neighbors v. To get a 1-round dMA protocol, with $O(\frac{n^2}{\alpha} \log n)$-bit certificates, and $O(\alpha \log n)$-bit communication, Merlin sends to u a specific certificate for each edge incident to u. That is, u gets a polynomial Φ_v for each $v \in S_u$, which equals (allegedly) to $\Psi_{uv}(i) = \sum_{t\in T} \Psi_{uv,t}(i) = \sum_{t\in T} \Psi_{S_u,t}(i) \cdot \Psi_{S_v,t}(i)$ on each $i \in \mathbb{F}_q$. In this case, v chooses i_v at random locally, and sends to u the value i_v in addition to the α evaluations $\Psi_{S_v,t}(i_v)$ for all $t \in [\alpha]$.

A particular application of Theorem 4 is the existence of a Merlin-Arthur protocol with both space and message complexities $\widetilde{O}(\sqrt{n})$. This contrasts with the following lower bound.

▶ **Proposition 5.** *Any proof-labeling scheme for triangle-freeness must have certificate size at least $n/e^{O(\sqrt{\log n})}$ bits.*

Proof idea. The lower bound graph construction for the BROADCAST-CONGESTED-CLIQUE model [9] obviously gives a lower bound to the weaker, BROADCAST-CONGEST model. This lower bound is based on a lower bound for multiparty communication complexity of disjointness [27], which also applies for the non-deterministic case. Finally, as noted in previous work on PLS [18, 6], a lower bound for non-deterministic communication complexity in the BROADCAST-CONGEST model implies a certificate-size lower bound for PLS. ◀

We can then conclude that, as opposed to the dMA protocol of Theorem 4, any PLS for triangle freeness must use almost-linear communication. Put differently, the trivial protocol of sending all the list of neighbors is almost optimal, even if non-determinism is used.

4 Distributed vs. Shared Randomness

In this section we compare the power of distributed interactive protocols using *shared* randomness (the nodes have access to a common source of random coins) with the power of protocols using *distributed* randomness (each node has access to a private source of random coins only) – in both cases, the outcomes of the random trials are public to Merlin.

Certifying Solutions to Optimization Problems

We consider optimization problems on graphs, such as finding a minimum dominating set, or a maximum independent set, and their weighted counterparts. Similar problems where previous studied in the context of non-interactive distributed verification [11]. In such a problem π, an admissible solution is a set S of nodes satisfying a set of constraints depending on π, and the quality of a solution S is measured by its weight $w(S) = \sum_{s\in S} w(s)$ where $w(s)$

is the weight of node s, given as input (where $w(s) = 1$ for every node s when considering only the cardinality of the solution). We assume that all weights are polynomial in the size n of the network. A set S is distributively encoded by a Boolean variable $x(v)$ at each node v, indicating whether the node is in S or not. We consider two distributed languages:

- The language Adm_π is composed of all configurations $(G, (w, x), \mathrm{id})$ such that x encodes an *admissible* solution for π in the weighted graph G (weights are assigned by w).
- The language $\mathsf{OptVal}_{\pi,k}$, for $k \geq 0$, is composed of all configurations (G, w, id) such that there exists an admissible solution for π of weight at most k (respectively, at least k) for the minimization (respectively, maximization) problem π.

This framework can easily be extended to study problems whose solutions are sets of edges.

▶ **Theorem 6.** *For any optimization problem π on graphs such that checking whether a given solution x is admissible can done by exchanging $O(\log \log n)$ bits between neighbors, there exists a Merlin-Arthur protocol for $\mathsf{OptVal}_{\pi,k}$, using $O(\log n)$ bits of shared randomness, with $O(\log n)$-bit certificates and $O(\log \log n)$-bit messages between nodes. In short, $\mathsf{OptVal}_{\pi,k} \in \mathsf{dMA}(\log n,\ \log \log n)$.*

Proof idea. A trivial starting point for protocols solving π is by having Merlin mark the nodes of the solution using the certificates. Computing the weight of the given solutions is a global task, and thus much harder in the distributed setting. To solve it, we aggregate the solution weight over a spanning tree. However, this still requires larger messages than desired; the crucial part in the proof is in using multi-party communication complexity protocol with a referee for the SumZero problem. Using this protocol, we can reduce the messages to their desired size. ◀

For the statement of Theorem 6, we assumed that all weights are polynomial in the size n of the network. If the weights are m-bit long, we can adapt the proof, and show that there exists a Merlin-Arthur protocol for $\mathsf{OptVal}_{\pi,k}$, using $O(\log(m + \log n))$ bits of shared randomness, with $O(\log n)$-bit certificates and $O(\log n)$-bit messages between nodes. In short, $\mathsf{OptVal}_{\pi,k} \in \mathsf{dMA}(\log n,\ \log n)$ even with weights exponential in n.

Certifying Coloring and Lucky Labeling

Similar arguments as the ones used to establish Theorem 6 allow us to verify specific optimization problems, for which checking that a solution is admissible is not easy. We exemplify this with the coloring problem, and its variant the *lucky labeling* problem [4, 8].

Checking that a given graph coloring is proper is a simple task, which can be solved by having each node broadcast its color. Here, we show that verifying a given c-coloring can be done using a dMA protocol with $O(\Delta \log \log c)$-bit certificates and $O(1)$ bits of communication in networks of maximum degree Δ. This stands in contrast to the trivial verification algorithm where the communication is of $O(\log c)$ bits. In the dMA protocol, Merlin provides every node v with the location $p(v, u)$ of a bit where the colors of u and v differ, for each of its neighbors u. A node v then checks with each neighbor u the fact that $p(v, u) = p(u, v)$, and at the same time sends to u the value of the corresponding bit. The Merlin step requires space $O(\Delta \log \log c)$. The Arthur step requires constant communication when shared randomness is available using the equality protocol.

▶ **Lemma 7.** *There is a Merlin-Arthur protocol for verifying a given c-coloring using $O(\log \log c)$ bits of shared randomness, certificates of size $O(\log \log c)$ bits, and constant communication complexity.*

We apply this remark to the so-called *lucky labeling.* Let $\ell : V \to \{1, \ldots, c\}$, and, for every node v of G, let $S(v) = \sum_{u \in N(v)} \ell(u)$. The labeling ℓ is *lucky* if, for every two adjacent nodes u and v, we have $S(u) \neq S(v)$. The lucky coloring number of a graph G, denoted by $\eta(G)$, is the least positive integer c such that G has a lucky labeling $\ell : V \to \{1, \ldots, c\}$. We refer to [4, 8] for properties of the lucky coloring number. In particular, it is conjectured that $\eta(G) \leq \chi(G)$, and it is known that $\eta(G) \leq \Delta^2 - \Delta + 1$, even for list lucky labeling.

Verifying a given graph coloring is trivial, even without labels or interaction. To verify an upper bound on the chromatic number χ of a graph, there is a simple PLS giving each node a color. The situation with lucky labeling is much more subtle: it is impossible to verify lucky labeling in a single round (this can be easily seen by considering different labelings on a short path). There is a simple PLS for verifying a given labeling is lucky, or for bounding η from above, which gives each node v the sum $S(v)$, and also labels for the latter case.

This PLS has label size and communication of $O(\log \Delta)$. Applying RPLS [15] gives $\sigma = O(\Delta \log \Delta)$ and $\gamma = O(\log \log \Delta)$, which can be reduced to $\gamma = O(1)$ using shared randomness. Here, we show an MA protocol using shared randomness with $\sigma = O(\Delta \log \log \Delta)$ and $\gamma = O(\log \log \Delta)$, establishing another trade-off between space and communication.

▶ **Theorem 8.** *For every $\lambda = O(n)$, there exists a Merlin-Arthur protocol for $\eta(G) \leq \lambda$, using $O(\log \log \Delta)$ bits of shared randomness, with $O(\Delta \log \log \Delta)$-bit certificates and $O(\log \log \Delta)$-bit messages between nodes. In short, lucky labeling is in* $\mathsf{dMA}(\Delta \log \log \Delta,\ \log \log \Delta)$.

Proof. Merlin sends to every node v its label $\ell(v)$ and the alleged sum $S(v)$ of the labels of its neighbors. For the verification, the nodes verify that the sums $S(v)$ constitute a proper coloring, using the protocol from Lemma 7. In addition, they use a multiparty protocol for the SumZero problem [26, 3], in order to verify $S(v) = \sum_{u \sim v} \ell(u)$. ◀

A similar protocol can be used for the problem of verifying that a given labeling is lucky.

A General Reduction Between Distributed and Shared Randomness

Interestingly, assuming distributed randomness does not limit the power of Arthur-Merlin protocols compared to *shared* randomness, up to a small additive factor in the certificate size. The same holds for Merlin-Arthur protocols, but solely up to one additional interaction between Arthur and Merlin. This result is not hard to achieve using a classical spanning-tree verification technique already applied in proof labeling schemes [22]. Yet, it both generalizes and simplifies the previous results on shared vs. distributed randomness [20].

▶ **Theorem 9.** *For any distributed language $\mathcal{L}$, and for any number $k \geq 1$ of interactions, and for any certificate size $\sigma \geq 0$, if $\mathcal{L} \in \mathsf{dAM}[k](\sigma, \gamma)$ (respectively, $\mathcal{L} \in \mathsf{dMA}[k](\sigma, \gamma)$) using $\rho(n)$ shared random bits, then $\mathcal{L} \in \mathsf{dAM}[k](\sigma + \log n + \rho, \gamma + \log n + \rho)$ (respectively, $\mathcal{L} \in \mathsf{dAM}[k+1](\sigma + \log n + \rho, \gamma + \log n + \rho)$) with distributed randomness.*

Proof idea. We simulate the shared randomness protocol by using the randomness of a single node as the shared randomness. Merlin disseminates the randomness to all nodes, and a spanning tree is used to verify that the disseminated random string is the desired one. ◀

Lower bound for shared randomness

For many verification problems, the number of random bits used by Arthur remains limited, typically $\rho(n) = O(\log n)$, which shows that, often, shared randomness does not add much power to Arthur-Merlin protocols. The next result states a lower bound on the certificate and message size in the case of Sym and $\overline{\mathsf{Sym}}$, when using shared randomness.

▶ **Theorem 10.** *Any Arthur-Merlin protocol for (non) symmetry must have certificate and message size* $\Omega(\log\log n)$*. In short,* $\mathsf{Sym}, \overline{\mathsf{Sym}} \notin \mathsf{dAM}(o(\log\log n), \infty) \cup \mathsf{dAM}(\infty, o(\log\log n))$*, even using shared randomness.*

Proof. In [17], a communication complexity variant of Arthur-Merlin protocols has been proposed. In this variant, Arthur consists of two parties, Alice and Bob, and the input is split between them: Alice holds x, Bob holds y, and they wish to decide whether the value of a specified function f with input x and y is equal to 1. At the beginning, Alice and Bob start by tossing some coins, then Merlin publishes a certificate, and finally Alice and Bob separately decide whether to accept (the acceptance/rejection criteria are the same as for the Arthur-Merlin protocols). The communication cost of the protocol is defined as the worst-case length of Merlin's certificates. At the end of the paper, the authors observe that, with respect to this variant of Arthur-Merlin protocols, any such protocol for Eq and for $\overline{\mathsf{Eq}}$ must have communication complexity $\Omega(\log\log n)$. We will now show that the existence of a dAM protocol with one interaction for the Sym (respectively, $\overline{\mathsf{Sym}}$) problem with certificate size $o(\log\log n)$ would imply a two-player Arthur-Merlin protocol for Eq (respectively, $\overline{\mathsf{Eq}}$) with communication complexity $o(\log\log n)$. The theorem thus follows.

Given two binary vectors $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$, recall that $\mathsf{Eq}(x, y) = 1$ if and only if $x_i = y_i$ for every i with $1 \leq i \leq n$ (for the sake of simplicity, we will assume that $x \neq \mathbf{0}$ and that $y \neq \mathbf{0}$, but our construction can be also adapted to the case in which $x = \mathbf{0}$ or $y = \mathbf{0}$). We now define a graph $G_{x,y}$ such that $\mathsf{Sym}(G_{x,y}) = 1$ if and only if $\mathsf{Eq}(x, y) = 1$ (and, hence, $\overline{\mathsf{Sym}}(G_{x,y}) = 1$ if and only if $\overline{\mathsf{Eq}}(x, y) = 1$). The graph includes $6n + 2$ nodes a, b, a_i, b_i, u_i, v_i, y_i, and z_i, for $1 \leq i \leq n$, and the following edges (see Fig. 2):

- (a, b), (a, a_i), for $1 \leq i \leq n$ such that $x_i = 1$, and (b, b_i), for $1 \leq i \leq n$ such that $y_i = 1$;
- (u_i, a_j) and (v_i, b_j), for $1 \leq i \leq j \leq n$;
- (u_i, u_j) and (v_i, v_j), for $1 \leq i < j \leq n$, and (u_i, y_j) and (v_i, z_j), for $1 \leq i, j \leq n$;
- (y_i, y_{i+1}) and (z_i, z_{i+1}), for $1 \leq i < n$, and (y_1, y_n) and (z_1, z_n).

Clearly, if $x = y$, then $\mathsf{Sym}(G_{x,y}) = 1$: indeed, we can simply map each a-node (respectively, u-node and y-node) to the corresponding b-node (respectively, v-node and z-node). On the other hand, because of the degree distribution of its nodes, any non-trivial automorphism of $G_{x,y}$ has to map the u-nodes to the corresponding v-nodes: this in turn implies that, because of their neighborhoods, each a node has to be mapped to the corresponding b-node. Hence, since the mapping is an automorphism, the neighborhood of node a and node b has to be the same: that is, $x = y$.

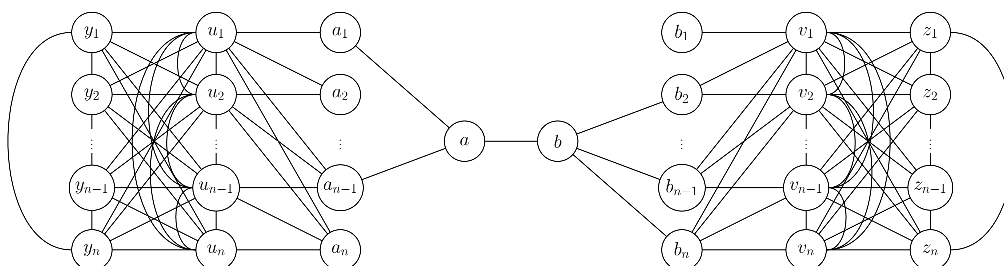


Figure 2 The graph used to reduce Eq (respectively, $\overline{\mathsf{Eq}}$) to Sym (respectively, $\overline{\mathsf{Sym}}$): in this case, $x_2 = x_n = y_1 = 0$.

Let us now suppose that there exists a dAM protocol $\mathcal{P}$ with one interaction for the Sym (respectively, $\overline{\mathsf{Sym}}$) problem which uses certificates of size $o(\log\log n)$. We now show how $\mathcal{P}$ can be used to design an Arthur-Merlin protocol for Eq (respectively, $\overline{\mathsf{Eq}}$) with communication complexity $o(\log\log n)$ (for the sake of brevity, we will show this statement for Sym and Eq: the proof for $\overline{\mathsf{Sym}}$ and $\overline{\mathsf{Eq}}$ is almost identical). Given x (respectively, y), Alice (respectively, Bob) can construct the (a,u,y)-subgraph (respectively, (b,v,z)-subgraph) of $G_{x,y}$: let $G^A_{x,y}$ (respectively, $G^B_{x,y}$) denote such subgraph. After having sent to Merlin the shared random string r, Alice and Bob waits for Merlin's certificate which is supposed to be formed by the two certificates π_a and π_b that nodes a and b would have received during the execution of $\mathcal{P}$ with random string r. By simulating $\mathcal{P}$ for every possible certificate assignment to the nodes of $G^A_{x,y}$ (respectively, $G^B_{x,y}$), Alice (respectively, Bob) can verify whether there exists an assignment that makes all the nodes of its corresponding subgraph accept: if this is the case, Alice (respectively, Bob) accepts. By definition, we have that if $x=y$, then, for any random string r there exist a certificate assignment to $G^A_{x,y}$ (respectively, $G^B_{x,y}$) and a certificate for Alice and Bob which make Alice and Bob accept. On the other hand, if $x\neq y$, then, for at least $2/3$ of all possible random strings, any certificate assignment to $G^A_{x,y}$ (respectively, $G^B_{x,y}$) and any certificate for Alice and Bob makes Alice and Bob reject. Since the size of the certificate for Alice and Bob is twice the size of the certificate size of $\mathcal{P}$, this implies that this protocol is an Arthur-Merlin protocol for Eq with communication complexity $o(\log\log n)$. This contradicts the lower bound observed in [17]: we have thus proved that $\mathsf{Sym},\overline{\mathsf{Sym}}\notin\mathsf{dAM}(o(\log\log n),\infty)$.

The above proof can be adapted in order to obtain a lower bound on the communication complexity of any dAM for the Sym and $\overline{\mathsf{Sym}}$ problems. Indeed, instead of asking Merlin for the certificates of the nodes a and b, Alice and Bob ask Merlin for the message transmitted on the edge (a,b). They then try to find a certificate assignment that suits this message. Hence, we have also shown that $\mathsf{Sym},\overline{\mathsf{Sym}}\notin\mathsf{dAM}(\infty,o(\log\log n))$ and the theorem follows. ◀

A result similar to previous theorem was proved in [20] in an ad-hoc manner, but only for the Sym problem and with respect to space complexity. The authors have recently reported to improve the lower bound from $\log\log n$ to $\log n$ [24].

5 Interactions vs. Space and Communication

In this section, we explore the power given to interactive protocols by allowing many interactions between Merlin and Arthur, in terms of both space and communication complexity.

Reducing the number of interactions

The following general result allows us to reduce the number of interactions between Arthur and Merlin, at the cost of increasing the certificate size and the communication cost of the protocol.

▶ **Theorem 11.** *For any two functions σ and γ,* $\mathsf{dMAM}(\sigma,\gamma)\subseteq\mathsf{dAM}(n\sigma^2,n\sigma\gamma)$.

Proof idea. We modify a dMAM protocol, so that Merlin gives all the certificates at the end, instead of at interactions 1 and 3. This results in a very low success probability, so we repeat the new protocol for $n\sigma$ times in parallel, and accept only if all occurrences are accepting. ◀

▶ **Corollary 12.** *For any two functions σ and γ,* $\mathsf{dAMAM}(\sigma,\gamma)\subseteq\mathsf{dAM}(n\sigma^2,n\sigma\gamma)$.

As a direct application of Theorem 11, we have that Sym admits a dAM protocol with one interaction and certificate size $O(n \log^2 n)$. This is a consequence of the theorem and of the existence of a dAM protocol with two interactions [20]. It is worth noting that this consequence of our general reduction result is only a log factor away from the "ad-hoc" result of [20] which establishes the existence of a dAM protocol with one interaction and certificate size $O(n \log n)$. Corollary 12 can be applied to a more recent result[24], which establishes the existence of a dAM protocol with three interactions and certificate size $O(\log n)$ for the non-isomorphism graph problem, in the case in which the nodes can communicate on both graphs. As a consequence of the corollary, we have that this problem admits a dAM protocol with one interaction and certificate size $O(n \log^2 n)$. As far as we know, this is the first dAM protocol with one interaction for this version of the non-isomorphism graph problem An interesting open question is whether such a protocol can exist also for the problem in which nodes can communicate only on one graph, while the other graph is locally given as input to the nodes themselves. For this latter problem, a dAM protocol with one interaction and certificate size $O(n \log n)$ was given [20], as well as an dAM protocol with a constant number of interactions and certificate size $O(\log n)$ [24]. Observe that the two applications of Theorem 11 and of Corollary 12 are obtained at the cost of an increase of the communication complexity by a factor $\widetilde{O}(n)$. We do not know if this linear increase of communication complexity can be avoided in general.

The Arthur-Merlin Hierarchy

We analyze the power of the Arthur-Merlin hierarchy. Recall that, for any $\sigma \geq 0$ and $\gamma \geq 0$, $\mathsf{dAMH}(\sigma, \gamma) = \bigcup_{k \geq 0} \mathsf{dAM}[k](\sigma, \gamma)$. We show that increasing the number of interactions cannot help much for reducing the certificate size to $o(n)$, even for languages defined on a very simple subclass of regular graphs, with 1-bit inputs, and admitting a locally checkable proof with $O(n)$-bit certificates.

▶ **Theorem 13.** *There exists a distributed language $\mathcal{L}$ on cycles, with 1-bit inputs, admitting a locally checkable proof with $O(n)$-bit certificates, and $O(n)$-bit messages, that is outside the Arthur-Merlin hierarchy with $o(n)$-bit certificates, even with messages of unbounded size, and even if the verifier performs an arbitrarily large constant number of rounds, whenever Arthur generates $\rho(n) = o(n)$ random bits at each node for each interaction with Merlin. In short, there exists a distributed language $\mathcal{L}$ on regular graphs satisfying $\mathcal{L} \in \Sigma_1\mathsf{LD}(O(n), O(n)) \setminus \mathsf{dAMH}(o(n), \infty)$.*

Proof idea. The main argument of the proof is that the number of transcripts of Arthur-Merlin protocol with $o(n)$-bit certificates, $o(n)$-bit random strings, and $k = O(1)$ interactions, is smaller than the number of distributed languages on n-node graphs, even with 1-bit input labels, and even on the ring. On the other hand, with $\Theta(n)$-bit certificates, all such languages can be decided by a locally checkable proof on the ring. ◀

We complete this section by showing that, in contrast to the previous theorem, every distributed language on regular graphs with $O(1)$-bit inputs has a locally checkable proof with $O(n)$-bit certificates, and a 2-round verifier.

▶ **Theorem 14.** *Every distributed language on d-regular graphs with $O(1)$-bit input labels belongs to $\Sigma_1\mathsf{LD}(\widetilde{O}(n), O(dn))$, with a verifier performing two rounds.*

Proof idea. To prove such a general claim, one must allow each node to study the structure and inputs of the whole graph. To this end, we apply the probabilistic method and show that there is a "balanced" assignment of information to the nodes, such that each node can use the information in its 1-neighborhood in order to reconstruct the entire graph structure. ◀

Since $\Sigma_1\mathsf{LD} \subseteq \mathsf{dAM} \cap \mathsf{dMA}$, an immediate corollary of this theorem is that every distributed language on d-regular graphs with $O(1)$-bit inputs belongs to $\mathsf{dAM}(\widetilde{O}(n), O(dn))$. Using a known RPLS protocol [15], we can also show that each such language belongs to $\mathsf{dMA}(\widetilde{O}(dn), O(\log n))$.

References

1. Scott Aaronson and Avi Wigderson. Algebrization: A New Barrier in Complexity Theory. *TOCT*, 1(1):2:1–2:54, 2009. doi:10.1145/1490270.1490272.
2. Amir Abboud, Aviad Rubinstein, and R. Ryan Williams. Distributed PCP Theorems for Hardness of Approximation in P. In *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS*, pages 25–36, 2017. doi:10.1109/FOCS.2017.12.
3. Daniel Apon, Jonathan Katz, and Alex J. Malozemoff. One-round multi-party communication complexity of distinguishing sums. *Theor. Comput. Sci.*, 501:101–108, 2013. doi:10.1016/j.tcs.2013.07.026.
4. Maria Axenovich, Jochen Harant, Jakub Przybylo, Roman Soták, Margit Voigt, and Jenny Weidelich. A note on adjacent vertex distinguishing colorings of graphs. *Discrete Applied Mathematics*, 205:1–7, 2016. doi:10.1016/j.dam.2015.12.005.
5. Alkida Balliu, Gianlorenzo D'Angelo, Pierre Fraigniaud, and Dennis Olivetti. What can be verified locally? *J. Comput. Syst. Sci.*, 97:106–120, 2018. doi:10.1016/j.jcss.2018.05.004.
6. Keren Censor-Hillel, Ami Paz, and Mor Perry. Approximate Proof-Labeling Schemes. In *Structural Information and Communication Complexity - 24th International Colloquium, SIROCCO*, pages 71–89, 2017. doi:10.1007/978-3-319-72050-0_5.
7. Yi-Jun Chang, Seth Pettie, and Hengjie Zhang. Distributed Triangle Detection via Expander Decomposition. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 821–840, 2019. doi:10.1137/1.9781611975482.51.
8. Sebastian Czerwinski, Jaroslaw Grytczuk, and Wiktor Zelazny. Lucky labelings of graphs. *Inf. Process. Lett.*, 109(18):1078–1081, 2009. doi:10.1016/j.ipl.2009.05.011.
9. Andrew Drucker, Fabian Kuhn, and Rotem Oshman. On the power of the congested clique model. In *ACM Symposium on Principles of Distributed Computing, PODC*, pages 367–376, 2014. doi:10.1145/2611462.2611493.
10. Laurent Feuilloley and Pierre Fraigniaud. Survey of Distributed Decision. *Bulletin of the EATCS*, 119, 2016. URL: http://eatcs.org/beatcs/index.php/beatcs/article/view/411.
11. Laurent Feuilloley, Pierre Fraigniaud, and Juho Hirvonen. A Hierarchy of Local Decision. In *43rd International Colloquium on Automata, Languages, and Programming, ICALP*, pages 118:1–118:15, 2016. doi:10.4230/LIPIcs.ICALP.2016.118.
12. Laurent Feuilloley, Pierre Fraigniaud, Juho Hirvonen, Ami Paz, and Mor Perry. Redundancy in Distributed Proofs. In *32nd International Symposium on Distributed Computing, DISC*, pages 24:1–24:18, 2018. doi:10.4230/LIPIcs.DISC.2018.24.
13. Pierre Fraigniaud, Mika Göös, Amos Korman, Merav Parter, and David Peleg. Randomized distributed decision. *Distributed Computing*, 27(6):419–434, 2014. doi:10.1007/s00446-014-0211-x.
14. Pierre Fraigniaud, Amos Korman, and David Peleg. Towards a complexity theory for local distributed computing. *J. ACM*, 60(5):35:1–35:26, 2013. doi:10.1145/2499228.
15. Pierre Fraigniaud, Boaz Patt-Shamir, and Mor Perry. Randomized proof-labeling schemes. *Distributed Computing*, 32(3):217–234, 2019.

16 John Gill. Computational Complexity of Probabilistic Turing Machines. *SIAM J. Comput.*, 6(4):675–695, 1977. doi:10.1137/0206049.

17 Mika Göös, Toniann Pitassi, and Thomas Watson. Zero-Information Protocols and Unambiguity in Arthur-Merlin Communication. *Algorithmica*, 76(3):684–719, 2016. doi:10.1007/s00453-015-0104-9.

18 Mika Göös and Jukka Suomela. Locally Checkable Proofs in Distributed Computing. *Theory of Computing*, 12(1):1–33, 2016. doi:10.4086/toc.2016.v012a019.

19 Karthik C. S., Bundit Laekhanukit, and Pasin Manurangsi. On the parameterized complexity of approximating dominating set. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC*, pages 1283–1296, 2018. doi:10.1145/3188745.3188896.

20 Gillat Kol, Rotem Oshman, and Raghuvansh R. Saxena. Interactive Distributed Proofs. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC*, pages 255–264, 2018. URL: https://dl.acm.org/citation.cfm?id=3212771.

21 Amos Korman and Shay Kutten. Distributed verification of minimum spanning trees. *Distributed Computing*, 20(4):253–266, 2007. doi:10.1007/s00446-007-0025-1.

22 Amos Korman, Shay Kutten, and David Peleg. Proof labeling schemes. *Distributed Computing*, 22(4):215–233, 2010. doi:10.1007/s00446-010-0095-3.

23 Eyal Kushilevitz and Noam Nisan. *Communication complexity*. Cambridge University Press, 1997.

24 Moni Naor, Merav Parter, and Eylon Yogev. The Power of Distributed Verifiers in Interactive Proofs. *CoRR*, abs/1812.10917, 2018. arXiv:1812.10917.

25 Moni Naor and Larry J. Stockmeyer. What Can be Computed Locally? *SIAM J. Comput.*, 24(6):1259–1277, 1995. doi:10.1137/S0097539793254571.

26 Noam Nisan. The communication complexity of threshold gates. In *Combinatorics, Paul Erdos is Eighty*, volume 1, pages 301–315, 1993.

27 Anup Rao and Amir Yehudayoff. Simplified Lower Bounds on the Multiparty Communication Complexity of Disjointness. In *30th Conference on Computational Complexity, CCC*, pages 88–101, 2015. doi:10.4230/LIPIcs.CCC.2015.88.

28 Larry J. Stockmeyer. The Polynomial-Time Hierarchy. *Theor. Comput. Sci.*, 3(1):1–22, 1976. doi:10.1016/0304-3975(76)90061-X.

The Capacity of Smartphone Peer-To-Peer Networks

Michael Dinitz
Johns Hopkins University, Baltimore, MD, United States
mdinitz@cs.jhu.edu

Magnús M. Halldórsson
Reykjavík University, Iceland
mmh@ru.is

Calvin Newport
Georgetown University, Washington, DC, United States
cnewport@cs.georgetown.edu

Alex Weaver
Georgetown University, Washington, DC, United States
aweaver@cs.georgetown.edu

Abstract

We study three capacity problems in the mobile telephone model, a network abstraction that models the peer-to-peer communication capabilities implemented in most commodity smartphone operating systems. The *capacity* of a network expresses how much sustained throughput can be maintained for a set of communication demands, and is therefore a fundamental bound on the usefulness of a network. Because of this importance, wireless network capacity has been active area of research for the last two decades.

The three capacity problems that we study differ in the structure of the communication demands. The first problem is pairwise capacity, where the demands are (source, destination) pairs. Pairwise capacity is one of the most classical definitions, as it was analyzed in the seminal paper of Gupta and Kumar on wireless network capacity. The second problem we study is broadcast capacity, in which a single source must deliver packets to all other nodes in the network. Finally, we turn our attention to all-to-all capacity, in which all nodes must deliver packets to all other nodes. In all three of these problems we characterize the optimal achievable throughput for any given network, and design algorithms which asymptotically match this performance. We also study these problems in networks generated randomly by a process introduced by Gupta and Kumar, and fully characterize their achievable throughput.

Interestingly, the techniques that we develop for all-to-all capacity also allow us to design a one-shot gossip algorithm that runs within a polylogarithmic factor of optimal in every graph. This largely resolves an open question from previous work on the one-shot gossip problem in this model.

2012 ACM Subject Classification Computing methodologies → Distributed algorithms; Networks → Network algorithms

Keywords and phrases Capacity, Wireless, Mobile Telephone, Throughput

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.14

Related Version A full version of this paper is available at `https://arxiv.org/abs/1908.01894`.

Funding *Michael Dinitz*: Supported in part by NSF award CCF-1535887.
Magnús M. Halldórsson: Supported in part by Icelandic Research Fund grant 174484.
Calvin Newport: Supported in part by NSF award CCF-1733842.
Alex Weaver: Supported in part by NSF award CCF-1733842.

© Michael Dinitz, Magnús M. Halldórsson, Calvin Newport, and Alex Weaver;
licensed under Creative Commons License CC-BY
33rd International Symposium on Distributed Computing (DISC 2019).
Editor: Jukka Suomela; Article No. 14; pp. 14:1–14:17

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

In this paper, we study the classical capacity problem in the mobile telephone model: an abstraction that models the peer-to-peer communication capabilities implemented in most commodity smartphone operating systems. The capacity of a network expresses how much sustained throughput can be maintained for a set of communication demands. We focus on three variations of the problem: *pairwise* capacity, in which nodes are divided into pairwise packet flows, *broadcast* capacity, in which a single source delivers packets to the whole network, and *all-to-all* capacity, in which all nodes deliver packets to the whole network.

For each variation we prove limits on the achievable throughput and analyze algorithms that match (or nearly match) these bounds. We study these results in both *arbitrary* networks and *random* networks generated with the process introduced by Gupta and Kumar in their seminal paper on wireless network capacity [19]. Finally, we deploy our new techniques to largely resolve an open question from [24] regarding optimal one-shot gossip in the mobile telephone model. Below we summarize the problems we study and the results we prove, interleaving the relevant related work.

The Mobile Telephone Model. The *mobile telephone model* (MTM), introduced by Ghaffari and Newport [14], modifies the well-studied *telephone model* of wired peer-to-peer networks (e.g., [10, 15, 4, 17, 9, 16]) to better capture the dynamics of standard smartphone peer-to-peer libraries. It is inspired, in particular, by the specific interfaces provided by Apple's Multipeer Connectivity Framework [2].

In this model, the network is modeled as an undirected graph $G = (V, E)$, where the nodes in V correspond to smartphones, and an edge $\{u, v\} \in E$ indicates the devices corresponding to u and v are close enough to enable a direct peer-to-peer radio link. Time proceeds in synchronous rounds. As in the original telephone model, in each round, each node can either attempt to initiate a connection (e.g., place a telephone call) with at most one of its neighbors, or wait to receive connection attempts. Unlike the original model, however, a waiting node can accept at most one incoming connection attempt. This difference is consequential, as many of the celebrated results of the original telephone model depend on the nodes' ability to accept an unbounded number of incoming connections (see [14, 6] for more discussion).[1] This restriction is motivated by the reality that standard smartphone peer-to-peer libraries limit the number of concurrent connections per device to a small constant (e.g., for Multipeer this limit is 8). Once connected, a pair of nodes can participate in a bounded amount of reliable communication (e.g., transfer a constant number of packets/rumors/tokens).

Finally, the mobile telephone model also allows each node to broadcast a small $O(\log n)$-bit advertisement to its neighbors at the start of each round before the connection decisions are made. Most existing smartphone peer-to-peer libraries implement this *scan-and-connect* architecture. Notice, the mobile telephone model is harder than the original telephone model due to its connection restrictions, but also easier due to the presence of advertisements. The results is that the two settings are formally incomparable: each requires its own strategies for solving key problems.

[1] This behavior is particularly evident in studying PUSH-PULL rumor spreading in the telephone model in a star network topology. This simple strategy performs well in this network due to the ability of the points of the star to simultaneously pull the rumor from the center. In the mobile telephone model, by contrast, any rumor spreading strategy would be fundamentally slower due to the necessity of the center to connect to the points one by one.

In recent years, several standard one-shot peer-to-peer problems have been studied in the MTM, including rumor spreading [14], load balancing [7], leader election [24], and gossip [24, 25]. This paper is the first to study ongoing communication in this setting.

The Capacity Problem. Capacity problems are parameterized with a network topology $G = (V, E)$, and a flow set F made up of pairs of the form (s, R) (each of which is a *flow*), where $s \in V$ indicates a source (sometimes called a sender), and $R \subset V$ indicates a set of destinations (receivers). For each flow $(s, R) \in F$, source s is tasked with routing an infinite sequence of packets to destinations in R. The throughput achieved by a given destination for a particular flow is the average number of packets it receives from that flow per round in the limit, and the overall throughput is the smallest throughput over all the destinations in all flows (see Section 2.2 for formal definitions). We study three different capacity problems, each defined by the different constraints they place on the flow set F.

Results: Pairwise Capacity. The pairwise capacity problem divides nodes into source and destination pairs in F, i.e., the given flows are between pairs of nodes rather than from a source to a general destination set. We begin with pairwise capacity as it was the primary focus of Gupta and Kumar's seminal paper on the capacity of the protocol and physical wireless network models [19]. They argued that it provides a useful assessment of a network's ability to handle concurrent communication.

We begin in Section 3.1 by tackling the following fundamental problem: given an arbitrary connected network topology graph $G = (V, E)$ and a flow set F that divides the nodes in V into sender and receiver pairs, is it possible to efficiently calculate a packet routing schedule that approximates the optimal achievable throughput? We answer this question in the affirmative by establishing a novel connection between pairwise capacity and the classical concurrent multi-commodity flow (MCF) problem. To do so, we first transform a given G and F into an instance of the MCF problem. We then apply an existing MCF approximation algorithm to generate a fractional flow that achieves a good approximation of the optimal flow in the network. Finally, we apply a novel rounding procedure to transform the fractional flow into a schedule. We prove that this resulting schedule provides a constant approximation of the optimal achievable throughput.

Inspired by Gupta and Kumar [19], in Section 3.2 we turn our attention to networks and flow pairings that are randomly generated using the process introduced in [19]. This process is parameterized with a network size $n \geq 2$ and communication radius $r > 0$. It randomly places the n nodes in a unit square and adds an edge between any pair of nodes within distance r. The source and destination pairs are also randomly generated.

For every given size n, we identify a *connectivity threshold* value $r_c(n) = \Theta(\sqrt{\log n/n})$, such that for any radius $r \leq r_c(n)$, with constant probability the network generated by the above process for n and r includes a source with no path to its destination – trivializing the optimal achievable throughput to 0. We then prove that for every radius r that is at least a sufficiently large constant factor larger than the threshold, there is a tight bound of $\Theta(r)$ on the optimal achievable throughput. These results fully characterize our algorithm from Section 3.1 in randomly generated networks.

Results: Broadcast Capacity. Broadcast capacity is another natural communication problem in which a single *source* node is provided an infinite sequence of packets to deliver to all other nodes in the network. Solutions to this problem would be useful, for example, in a scenario where a large file is being distributed in a peer-to-peer network of smartphone users

in a setting without infrastructure. In Section 4.1 we study the optimal achievable throughput for this problem in arbitrary connected graphs. To do so, we connect the scheduling of broadcast packets to existing results on *graph toughness*, a metric that captures a graph's resilience to disconnection that was introduced by Chvátal [5] in the context of studying Hamiltonian paths.

In more detail, a graph G has a k-tree if there exists a spanning tree of G with maximum degree k. Let $d(G)$ be the smallest k such that G has a k-tree. This tree is also called a minimum degree spanning tree (MDST) of G. Building on a result of Win [29] that relates k-trees to toughness, we prove that for any given G with $d(G) > 3$, there exists a subset S of nodes such that removing S from G partitions the graph into at least $(d(G) - 2)|S|$ connected components.

As we formalize in Section 4.1, because each node in S can connect to at most one component per round (due to the connection restrictions of the mobile telephone model), $\Omega(d(G))$ rounds are required to spread each packet to all components, implying that no schedule achieves throughput better than $O(1/d(G))$.

In Section 4.2, we prove this bound tight by exhibiting a matching algorithm. The algorithm begins by constructing a k-tree T with $k \in \Theta(d(G))$ using existing techniques; e.g., [11, 8]. It then edge colors T and uses the colors as the foundation for a TDMA schedule of length $\Theta(k)$ that allows nodes to simulate the more powerful CONGEST model in which each node can connect with every neighbor in a round. In the CONGEST model, a basic pipelined broadcast provides constant throughput. When combined with the simulation cost the achieved throughput is an asymptotically optimal $\Omega(1/d(G))$.

It is straightforward for a centralized algorithm to calculate this schedule in polynomial time, but in some cases a pre-computation of this type might be impractical, or require too high of a setup cost.[2] With this in mind, we also provide a distributed version of this algorithm that converges to $\Omega(1/(d(G) + \log n))$ throughput in $\tilde{O}(D(T)d(G) + \sqrt{n})$ rounds, where $D(T)$ is the diameter of the spanning tree and $\tilde{O}$ hides polylog(n) factors. The algorithm further converges to an optimal $\Omega(1/d(G))$ throughput after no more than $O(n^2)$ total rounds – providing a trade-off between setup cost and eventual optimality.

Finally, in Section 4.3, we study the performance of our algorithm in networks generated randomly using the Gupta and Kumar process summarized above. We prove that for any communication radius sufficiently larger than the connectivity threshold, the network is likely to include an $O(1)$-tree, enabling our algorithms to converge to constant throughput. This result indicates that in evenly distributed network deployments the mobile telephone model is well-suited for high performance broadcast.

Results: All-to-All Capacity. All-to-all capacity generalizes broadcast capacity such that now *every* node is provided an infinite sequence of packets it must deliver to the entire network. Solutions to this problem would be useful, for example, in a local multiplayer gaming scenario in which each player needs to keep track of the evolving status of all other players connected in a peer-to-peer network.

Clearly, n separate instances of our broadcast algorithm from Section 4.2, one for each of the n nodes as the broadcast source, can be interleaved with a round robin schedule to produce $\Omega(1/(n \cdot d(G)))$ throughput. In Section 5, we draw on the same graph theory

[2] In the mobile telephone model, all nodes can learn the entire network topology in $O(n^2)$ rounds and then run a centralized algorithm locally to determine their routing behavior. Though this setup cost is averaged out when calculating throughput in the limit, it might be desirable to minimize it in practice.

connections as before to prove that this result is tight for all-to-all capacity. We then provide a less heavy-handed distributed algorithm for achieving this throughput. Instead of interleaving n different broadcast instances, it executes distinct instances of all-to-all gossip, one for each packet number, using a flood-based strategy on a low degree spanning tree. Finally, we apply the random graph analysis from Section 4.3 to establish that for sufficiently large communication radius, with high probability, the randomly generated graph supports $\Omega(1/n)$-throughput, which is trivially optimal in the sense that a receiver can receive at most one new packet per round in our model.

New Results on One-Shot Gossip. As we detail in Section 5.1, our results on all-to-all capacity imply new lower and upper bounds on one-shot gossip in the mobile telephone model. From the lower bound perspective, they imply that gossiping in graph G in the mobile telephone model requires $\Omega(n \cdot d(G))$ rounds. From the upper bound perspective, when we carefully account for the costs of our routing algorithm applied to spreading only a single packet from each source, we solve the one-shot problem with high probability in the following number of rounds:

$$O((D + \sqrt{n})\text{polylog}(n) + n(d(G) + \log n)) = \tilde{O}(d(G) \cdot n),$$

where D is the diameter of G. This algorithm is asymptotically optimal in any graph with $d(G) \in \Omega(\log n)$ and $D \in O(n/\log^x n)$ (where x is the constant from the polylog in the MDST construction time), which describes a large family of graphs. For all other graphs the solution is at most a polylog factor slower than optimal. This is the first known gossip solution to be optimal, or within log factors of optimal, in *all* graphs, largely answering a challenge presented by [24].

Motivation. Smartphone operating systems include increasingly robust support for opportunistic device-to-device communication through standards such as Apple's Multipeer Connectivity Framework [2], Bluetooth LE [18], and WiFi Direct [3]. Though the original motivation for these links was to support information transfer among a small number of nearby phones, researchers are beginning to explore their potential to enable large-scale peer-to-peer networks. Recent work, for example, uses smartphone peer-to-peer networking to provide disaster response [28, 26, 21], circumvent censorship [12], extend internet access [1, 13], support local multiplayer gaming [23] and improve classroom interaction [20].

It remains largely an open question whether or not it will be possible to build large-scale network systems on top of smartphone peer-to-peer links. As originally argued by Gupta and Kumar [19], bounds for capacity problems can help resolve such questions for a given network model by establishing the limit to their ability to handle ongoing and concurrent communication. The results in this paper, as well as the novel technical tools developed to prove them, can therefore help resolve this critical question concerning this important emerging network setting.

2 Preliminaries

Here we define our model, the problem we study, and some useful mathematical tools and definitions. Due to space constraints, this version of the paper omits most proofs. All technical details can be found in the full version.

2.1 Model

The mobile telephone model describes a smartphone peer-to-peer network topology as an undirected graph $G = (V, E)$. The nodes in V correspond to the smartphone devices, and an edge $\{u, v\} \in E$ implies that the devices corresponding to u and v are within range to establish a direct peer-to-peer radio link. We use $n = |V|$ to indicate the network size.

Executions proceed in synchronous rounds labeled $1, 2, ...$, and we assume all nodes start during round 1. At the beginning of each round, each node $u \in V$ selects an *advertisement* of size at most $O(\log n)$ bits to broadcast to its neighbors $N(u)$ in G. After the advertisement broadcasts, each node u can either send a connection invitation to at most one neighbor, or wait to receive invitations. A node receiving invitations can accept at most one, forming a reliable pairwise connection. It follows from these constraints that the set of connections in a given round forms a matching.

Once connected, a pair of nodes can perform a bounded amount of reliable communication. For the capacity problems studied in this paper, we assume that a pair of connected nodes can transfer at most one packet over the connection in a given round. We treat these packets as black boxes that can only be delivered in this manner (e.g., you cannot break a packet into pieces, or attempt to deliver it using advertisement bits).

We assume when running a distributed algorithm in this model that each computational process (also called a *node*) is provided a unique ID that can fit into its advertisement and an estimate of the network size. It is provided no other *a priori* information about the network topology, though any such node can easily learn its local neighborhood in a single round if all nodes advertise their ID.

2.2 Problem

In this paper we measure capacity as the achievable throughput for various combinations of packet flow and network types. We begin by providing a general definition of *throughput* that applies to all settings we study. This definition makes use of an object we call a *flow set*, which is a set $F = \{(s_i, R_i) : 1 \leq i \leq k\}$ (for some $k \geq 1$) where each $s_i \in V$ and $R_i \subseteq V$ (for node set V). For a given flow set F, each $(s_i, R_i) \in F$ describes a packet flow of *type* i; i.e., source s_i is tasked with sending packets to all the destinations in set R_i. We refer to the packets from s_i as i*-packets.*

A *schedule* for a given G and F describes a movement of packets through the flows defined by F. Formally, a schedule is an infinite sequence of directed matchings, $M_1, M_2, ...$ on G, such that the edges in each M_t are labelled by packets, where we define a packet as a pair (i, j) with $i \in [|F|]$ and $j \in \mathbb{N}$ (i.e., (i, j) is the j'th packet of type i). We require that the packet labels for a schedule satisfy the property that if edge (u, v) in M_t is labelled with packet $p = (i, j)$, then there is a path in $\bigcup_{l<t} M_l$ from s_i to u where all edges on the path are labelled with p. (It is easy to see by induction that this corresponds precisely to the intuitive notion of packets moving through a mobile telephone network). We say that a packet p is *received* by a node u in round r if there is an edge $(v, u) \in M_r$ which is labelled p. A packet (i, j) is *delivered* by round r if every $x \in R_i$ receives it in some round t with $t \leq r$.

Given a schedule S for a graph G and flow set F, we can define the throughput achieved by the slowest rate, indicated in packets per round, at which any of the flows in F are satisfied in the limit. Formally:

▶ **Definition 2.1.** *Fix a schedule S defined with respect to network topology graph $G = (V, E)$ and flow set F. We say S* achieves throughput t *with respect to G and F, if there exists a* convergence round $r_0 \geq 1$*, such that for every $r \geq r_0$ and every packet type i:*

$$\left(\frac{del_i(r)}{r}\right) \geq t,$$

where $del_i(r)$ is the largest j such that for every $l \leq j$, packet (i, l) has been delivered by round r.

The above definition of throughput concerns performance in the limit, since r_0 can be arbitrarily large. In some cases, though, we might also be concerned with how quickly we achieve this limit. Our notion of convergence round allows us to quantify this, so we will provide bounds on the convergence round where relevant.

Many of the results in this paper concern algorithms that produce schedules. Our centralized algorithms take G and F as input and efficiently produce a compact description of an infinite schedule (i.e., an infinitely repeatable finite schedule). Our distributed algorithms assume a computational process running at each node in G, and for each $(s_i, R_i) \in F$, the source s_i is provided an infinite sequence of packets to deliver to R_i. An execution of such a distributed algorithm might contain communication other than the *flow packets* provided as input; e.g., the algorithm might distributedly (in the mobile telephone model) compute a routing structure to coordinate efficient packet communication. However, a unique schedule can be extracted from each such execution by considering only communication corresponding to the flow packets.

While our definition of throughput is for schedules and not algorithms, we will say that an algorithm *achieves* throughput α if it results in a schedule that achieves throughput α.

In the sections that follow, we consider three different types of capacity: *pairwise*, *broadcast*, and *all-to-all*. Each capacity type can be formalized as a set of constraints on the allowable flow sets. For each capacity type we study achievable throughput with respect to both *arbitrary* and *random* network topology graphs. In the arbitrary case, the only constraints on the graph is that it is connected. For the random case, we must describe a process for randomly generating the graph. To do so, we use the approach introduced for this purpose by Gupta and Kumar [19]: randomly place nodes in a unit square, and then add an edge between all pairs within some fixed radius. Formally:

▶ **Definition 2.2.** *For a given real value* radius r*, $0 < r \leq 1$, and network size $n \geq 1$, the* $\text{GK}(n, r)$ network generation process *randomly generates a network topology $G = (V, E)$ as follows:*

1. *Let $V = \{u_1, u_2, ..., u_n\}$. Place each of the n nodes in V uniformly at random in a unit square in the Euclidean plane.*
2. *Let $E = \{(u_i, u_j) : d(u_i, u_j) \leq r\}$, where d is the Euclidean distance metric.*

We will use the notation $G \sim GK(n, r)$ to denote that G is a random graph generated by the $GK(n, r)$ process. When studying a specific definition of capacity with respect to a network randomly generated with the GK process, it is necessary to specify how the flow set is generated. Because these details differ for each of the three capacity definitions, we defer their discussion to their relevant sections.

2.3 Mathematical Preliminaries

We begin with some basic definitions. Fix some connected undirected graph $G = (V, E)$. We define $c(G)$ to be the number of components in G. In a slight abuse of notation, we define $G \setminus S$, for $S \subseteq V$, to be the graph defined when we remove from G the nodes in S and their

adjacent edges. For a fixed integer $k > 1$, we say G *has a k-tree* if there exists a spanning tree in G with maximum degree k. Finally, let $d(G)$ be the smallest k such that G has a k-tree. That is, $d(G)$ describes the maximum degree of the *minimum degree spanning tree* (MDST) in G.

Some of our results will use the following simple corollary of a theorem of Win [29]. The proof, which utilizes the notion of *graph toughness* [29], can be found in the full version.

▶ **Theorem 2.3.** *Fix an undirected graph $G = (V, E)$ and degree $k \geq 3$. If $d(G) > k$, then there exists a non-empty subset of nodes $S \subset V$ such that there are more than $c(G \setminus S) > (k-2) \cdot |S|$.*

3 Pairwise Capacity

In their seminal paper [19], Gupta and Kumar approached the question of network capacity by considering the maximum throughput achievable for a collection of disjoint pairwise flows, each consisting of a single source and destination. They studied achievable capacity in both *arbitrary* networks as well as *random* networks. In this section, we apply this approach to the mobile telephone model.

To do so, we formalize the *pairwise capacity* problem as the following constraint on the allowable flow sets (see Section 2.2): for every pair $(s_i, R_i) \in F$, it must be the case that $R_i = \{x\}$ (i.e., $|R| = 1$), and neither s nor x shows up in any other pair in F.

3.1 Arbitrary Networks

We begin by designing algorithms that (approximate) the maximum achievable throughput in an arbitrary network. For now we will not focus on the convergence time, since our definition of capacity applies in the limit, so we describe the following as a centralized algorithm (the time required for each node to gather the full graph topology and run this algorithm locally to generate an optimal routing schedule is smoothed out over time). But as usual when considering centralized algorithms, we will care about the running time.

Formally, we define the Pairwise Capacity problem to be the optimization problem where we are given a graph $G = (V, E)$ and a pairwise flow set F, and are asked to output a description of an (infinite) schedule which maximizes the throughput. Our algorithm will in particular output a finite schedule which is infinitely repeated. Our approach is to establish a strong connection between multi-commodity flow and optimal schedules, and then apply existing flow solutions as a step toward generating a near optimal solution for the current network. In other words, we give an approximation algorithm for Pairwise Capacity via a reduction to a multi-commodity flow problem.

▶ **Theorem 3.1.** *There is a (centralized) algorithm for Pairwise Capacity that achieves throughput which is a $(3/2 + \epsilon)$-approximation of the optimal throughput, for any $\epsilon > 0$. The convergence time is $n^{O(1)}\epsilon^{-2}$ and the running time is $n^{O(1)}\epsilon^{-1}$.*

Multi-Commodity Flow. In the *maximum concurrent multi-commodity flow (MCMF)* problem, we are given a triple (D, M, cap), where $D = (V_D, E_D)$ is a digraph, M is collection $M \subseteq V_D \times V_D$ of node-pairs (each representing a *commodity*), and $cap : E_D \to \mathbb{R}_0^+$ are flow capacities on the edges. Let $K = |M|$ be the number of commodities. The output is a collection $f = (f_1, f_2, \ldots, f_K)$ of flows satisfying conservation and capacity constraints. Namely, for each flow f_i and for each vertex $v \in G$ where $v \notin \{s_i, t_i\}$, the flow into a node equals the flow going out: $\sum_{e=(u,v)\in E_D} f_i(e) = \sum_{e'=(v,w)\in E_D} f_i(e')$. Also, the flow

through each edge is upper bounded by its capacity: $f(e) = \sum_{i=1}^{K} f_i(e) \leq cap(e)$. Let $v(f_i) = \sum_{w,e=(s_i,w)\in E_D} f_i(e)$ be the *value* of flow i, or the total flow of commodity i leaving its source. The value of the total flow f is $v(f) = \min_{i=1}^{K} v(f_i)$, and our goal is to maximize $v(f)$. We refer to f as an *MCMF flow* and the constituent commodity flows as *subflows*.

The MCMF problem can be solved in polynomial-time by linear programming. There are also combinatorial approximation schemes known, and our version of the problem can be approximated within a $(1+\epsilon)$-factor in time $\tilde{O}((m+K)n/\epsilon^2)$ [22].

We first show how to round an MCMF flow to use less precision while limiting the loss of value. We say that a MCMF flow is *ϕ-rounded* if the flow of each commodity on each edge is an integer multiple of $1/\phi$: $\lfloor f_i(e)\cdot\phi \rfloor = f_i(e)\cdot\phi$, for all i, and all edges e. We show how to produce a rounded flow of nearly the same value.

▶ **Lemma 3.2.** *Let f be a MCMF flow and ϕ be a number. There is a rounding of f to a ϕ-rounded flow f' with value at least $v(f') \geq v(f)(1 - Km/\phi)$, and it can be generated in polynomial time.*

Proof. We focus on each subflow f_i. By standard techniques, each subflow f_i can be decomposed into a collection of paths $P_1, \ldots, P_s$ and values $\alpha_1, \ldots, \alpha_s$, with $s \leq m = |E|$, such that $f_i(e) = \sum_{j,P_j\ni e} \alpha_j$ for each edge e. Let $\alpha'_j = \lfloor \alpha_j \cdot \phi \rfloor/\phi$, for each j, and observe that $\alpha'_j \geq \alpha_j - 1/\phi$. We form the ϕ-rounded flow f' by $f'_i(e) = \sum_{j,P_j\ni e} \alpha'_j$, for each edge e. It is easily verified that conservation and capacity constraints are satisfied. By the bound on α', it follows that the value of the rounded flow is bounded from below by $v(f'_i) \geq v(f_i) - s/\phi \geq v(f_i) - m/\phi$. The value of each flow is trivially bounded from below by $v(f_i) \geq 1/K$ (which is achieved by sending $1/K$ of each commodity flow along a single path). Thus, $v(f'_i) \geq (1 - Km/\phi)v(f_i)$. ◀

We now turn to the reduction of Pairwise Capacity to MCMF. Given $G = (V, E)$ and F, along with a parameter τ, we form the flow network $\mathcal{D}_\tau = (D, M, cap_\tau)$ as follows. The undirected graph $G = (V, E)$ is turned into a digraph $D = (V_D, E_D)$ with two copies v^{in}, v^{out} of each vertex: $V_D = \{v^{in}, v^{out} : v \in V\}$ and edges $E_D = \{(u^{out}, v^{in}) : uv \in E\} \cup \{(v^{in}, v^{out}) : v \in V\}$. The source/destination pairs carry over: $M = \{(s^{in}, t^{out}) : (s, \{t\}) \in F\}$. Finally, capacities of edges in E_D are $cap_\tau(u_{out}, v_{in}) = \infty$ and $cap_\tau(v_{in}, v_{out}) = 1 + t_v \cdot \tau/2$, where t_v is the number of source/destination pairs in F in which v occurs. Observe that there is a one-to-one correspondence between simple paths in G and in D (modulo the in/out version of the start/end node).

▶ **Lemma 3.3.** *The throughput of any schedule on (G, F) is at most $\tau^*/2$, where τ^* is the largest value such that $\mathcal{D}_{\tau^*}$ has MCMF flow of value τ^*.*

Proof. Let $\mathcal{A}$ be a mobile telephone schedule and let T be its throughput. We want to show that $\mathcal{D}_{2T}$ has MCMF flow of value $2T$; this is sufficient to imply the lemma. We assume that packets flow along simple paths, and we achieve that by eliminating loops from paths, if necessary. By the throughput definition, there is a round $r_0 = r^{\mathcal{A},T}$ such that for every round $r \geq r_0$ and every source/destination pair i, the number of i-packets delivered by round r is at least $T \cdot r$. Let X_i be the first Tr_0 i-packets delivered (necessarily by round r_0), for each type i, and let $X = \cup_i X_i$. For each edge $e = uv$ and pair i, let $q_i(u, v)$ be the number of packets in X_i that passed through e, from u to v. Also, for a vertex v, let $a_i(v)$ denote the number of i-packets originating at v, i.e., $a_i(v) = Tr_0$ if $v = s_i$ and $a_i(v) = 0$ otherwise. Similarly, let $b_i(v)$ be the number of i-packets with v as its destination. Finally, let $q_i(v)$ be the number of packets in X_i that flow through v, but did not originate or terminate at v, and observe that $q_i(v) = \sum_{w,vw\in E} q_i(v, w) - a_i(v) = \sum_{u,uv\in E} q_i(u, v) - b_i(v)$.

Define the collection $f = (f_1, f_2, \ldots, f_K)$ of functions where for each i, $f_i(u_{out}, v_{in}) = 2q_i(u,v)/r_0$, for each edge $e = uv \in E$, and $f_i(v_{in}, v_{out}) = 2(q_i(v) + a_i(v) + b_i(v))/r_0$, for each vertex in V. Observe that the flow $f_i(u_{out}, v_{in})$ corresponds to twice the number of i-packets going from u to v (scaled by factor $1/r_0$). The flow $f_i(v_{in}, v_{out})$ from v_{in} to v_{out} corresponds to the number of packets in X_i coming into v plus the number of those going out of v (scaled by factor $1/r_0$), counting those that go through v twice, but those originating or terminating at v only *once*. We claim that f is a valid MCMF flow in $\mathcal{D}_{2T}$ of value $2T$, which implies the lemma. Let $f_i^a(v) = 2a_i(v)/r_0$ ($f_i^b(v) = 2b_i(v)/r_0$) be the amount of type-i flow originating (terminating) at v, respectively.

First, to verify flow conservation at nodes, consider a type i, and observe first that all packets in X start at the source s_i and end at the destination t_i.

$$f_i(v_{in}, v_{out}) = \frac{2(q_i(v) + a_i(v) + b_i(v))}{r_0} = \frac{2a_i(v)}{r_0} + \sum_{u, uv \in E} \frac{2q_i(u,v)}{r_0}$$
$$= f_i^a(v) + \sum_{u,(u_{out}, v_{in}) \in E_D} f_i(u_{out}, v_{in}) \ .$$

That is, the flow from each node v_{in} equals the flow coming in plus the flow generated at the node (noting also that no flow terminates at the node). Similarly, the flow into v_{out} equals the flow terminating at the node plus the node going out:

$$f_i(v_{in}, v_{out}) = \frac{2(q_i(v) + a_i(v) + b_i(v))}{r_0} = \frac{2b_i(v)}{r_0} + \sum_{w, vw \in E} \frac{2q_i(v,w)}{r_0}$$
$$= f_i^b(v) + \sum_{w,(v_{out}, w_{in}) \in E_D} f_i(v_{out}, w_{in}) \ .$$

Second, to verify capacity constraints, observe that if $q(v) = \sum_i q_i(v)$ is the number of packets that flow through node v, then

$$2q(v) + \sum_i (a_i(v) + b_i(v)) \le r_0 \ ,$$

since v needs to handle flowing-through packets in two separate rounds and it can only process a single packet in a round. Thus, the flow through (v_{in}, v_{out}) is bounded by

$$f(v_{in}, v_{out}) = \frac{2}{r_0}(q(v) + \sum_i (a_i(v) + b_i(v))) = \frac{1}{r_0}(2q(v) + \sum_i (a_i(v) + b_i(v))) + t_v T \le 1 + t_v T \ ,$$

satisfying the capacity constraints.

Finally, it follows directly from the definition of f_i^a (or f_i^b) that the flow value is $2T$. ◀

To prove Theorem 3.1 we need to introduce edge multicoloring.

▶ **Definition 3.4.** *Given a graph $G = (V, E)$ and a* color requirement *$r(e) \in \mathbb{N}$ for each edge $e \in E$. An* edge multicoloring *of (G, r) is a function $\pi : E \to 2^{\mathbb{N}}$ that satisfies the following: a) if $e_1, e_2 \in E$ are adjacent then $\pi(e_1) \cap \pi(e_2) = \emptyset$, and b) $|\pi(e)| \ge r(e)$, for each edge $e \in E$. The number of colors used is $|\cup_e \pi(e)|$, the size of the support for π.*

We shall use the follow result on edge multicolorings.

▶ **Theorem 3.5** (Shannon [27]). *Given a graph $G = (V, E)$ and a* color requirement *$r(e) \in \mathbb{N}$ for each edge $e \in E$, there is a polynomial-time algorithm that edge multicolors (G, r) using at most $3\Delta_r(v)/2$ colors, where $\Delta_r(v) = \sum_{e \ni v} r(e)$.*

We can now prove Theorem 3.1.

Proof of Theorem 3.1. Let (G, F) be a given Pairwise Capacity instance and let $\epsilon > 0$. We perform binary search to find a value τ such that: a) An $1 + \epsilon/4$-approximate MCMF algorithm produces flow f of value at least $\tau(1 - \epsilon/4)$ on $\mathcal{D}_\tau$, and b) The same does not hold for $\tau(1 + \epsilon/4)$. The resulting flow $f = (f_1, \ldots, f_K)$ is then of value at least $\tau^*(1 - \epsilon/4)/(1 + \epsilon/4) \geq \tau^*(1 - \epsilon/4)^2 \geq \tau^*(1 - 2\epsilon/4)$. Recall that K is the number of commodities, and so $K = |F|$.

Let $N = 4\epsilon^{-1}Km$. We apply Lemma 3.2 to create from f an N-rounded flow $f' = (f'_1, \ldots, f'_K)$. By Lemma 3.2, this decreases the flow value by a factor of at most $1 - Km/N = 1 - \epsilon/4$, i.e., $v(f') \geq (1 - \epsilon/4)v(f) \geq (1 - 3\epsilon/4)\tau^*$.

We then form an edge multicoloring instance on G as follows. Each edge e requires $r(e)$ colors, where $r(e) = \sum_i r_i(e)$ and $r_i(e) = f'_i(e) \cdot N$. The weighted degree of each node v is then $d_r(v) = \sum_{e \ni v} r(e) = N \sum_{e \ni v} f'(e) \leq N \sum_{e \ni v} f(e) = \sum_{e=(v,u) \in E_D} f(e) + \sum_{e'=(w,v) \in E_D} f(e) \leq 2N$, by node capacity constraints. We apply the algorithmic version of Shannon's Theorem 3.5 to edge multicolor (G, r) with at most $3N$ colors. This induces an initial schedule of length $3N$, which is then repeated as needed. Within each $3N$ rounds, $\sum_{v, e=(s_i, v)} r_i(e) = N \cdot v(f'_i)$ i-packets depart from its source s_i.

Let $r_0 = \frac{4n}{\epsilon}(3N)$. Consider the situation after round $r \geq r_0$. Observe that each packet is forwarded at least once during each $3N$ rounds, and thus it is delivered within $n(3N)$ rounds after it is transmitted from its source, since each path used is simple. Thus, the total number of type-i packets that remain in the system in the end is at most a $\epsilon/4$-fraction of the delivered packets. Averaged over the r rounds gives throughput of

$$\frac{N \cdot v(f'_i)}{3N} \cdot (1 - \epsilon/4) = \frac{1}{3} v(f'_i) \cdot (1 - \epsilon/4) .$$

Hence, the throughput achieved is at least

$$T \geq \frac{1}{3} v(f')(1 - \epsilon/4) \geq \frac{1}{3} v(f)(1 - \epsilon/4) \geq \frac{1}{3} \tau^*(1 - \epsilon) . \quad (1)$$

By Lemma 3.3, the throughput is then $3/2 + \epsilon$-approximation of optimal.

The computation performed is dominated by the application of Shannon's algorithm, which runs in time $O((\Delta_r + n)\hat{m})$, where $\hat{m}$ is the number of multiedges and $\Delta_r \leq 2N$ is the maximum weighted degree. Here, $\hat{m} = \sum_e q(e) = N \sum_e \sum_i f_i(e) \leq N \cdot m$. Hence, the number of computational steps is at most $O(mN^2) = O(m^3K^2\epsilon^{-2})$. The convergence time is $r_0 = \frac{4n}{\epsilon}(3N) = O(nmK\epsilon^{-2})$. ◀

We note that the factor $3/2$ cannot be avoided in a reduction to flow. Consider the graph G on six vertices $V = \{s_i, t_i : i = 0, 1, 2\}$ and edges $\{s_i t_{i'}, t_i t_{i'} : i = 0, 1, 2, i' = i - 1 \bmod 3\}$. The optimal throughput is $1/3$, with respect to $F = \{(s_i, t_i) : i = 0, 1, 2\}$. This corresponds to the directed graph D on nine nodes: $\{s_i, t_i^{in}, t_i^{out} : i = 0, 1, 2\}$ and edges $\{(s_i, t_{i'}^{in}), (t_{i'}^{out}, t_i^{in}), (t_i^{in}, t_i^{out}) : i = 0, 1, 2, i' = i - 1 \bmod 3\}$, and three subflows: $M = \{s_i, t_i^{out} : i = 0, 1, 2\}$. Then, $\mathcal{D}_1 = (D, M, cap_1)$, where $cap_1(t_i^{in}, t_i^{out}) = 2$, has flow of value 1.

3.2 Random Networks

We now consider achievable throughput for the pairwise capacity problem in networks randomly generated with the GK process defined in Section 2.2. Following the lead of the original Gupta and Kumar capacity paper [19], we assume the flow sets are also randomly

generated with uniform randomness and contain all the nodes (i.e., every node shows up as a source or destination). A minor technical consequence of this definition is that it requires us to constrain our attention to even network sizes.

We begin in Section 3.2.1 by identifying a threshold value for the radius r below which the randomly generated network is likely to be disconnected, trivializing the achievable throughput to 0. In Sections 3.2.2 and 3.2.3, we then prove that for any radius value r that is at least a sufficiently large constant factor greater than the threshold, with high probability in n, the optimal achievable throughput is in $\Theta(r)$.

3.2.1 Connectivity Threshold

When analyzing networks and flows generated by the $GK(n, r)$ network generation process, we must consider the radius parameter r. If r is too small, then we expect a network in which some sources are disconnected from their corresponding destinations, making the best achievable throughput trivially 0. Here we study a *connectivity threshold* value $r_c(n) = \sqrt{\frac{\alpha \log n}{n}}$, defined with respect to a network size n and a constant fraction α. We prove that for any $r \leq r_c(n)$, with probability at least $1/2$, given a network generated by $G(n, k)$ and a random pairwise flow set F, there exists at least one pair in F that is disconnected.

▶ **Theorem 3.6.** *There is some constant $\alpha > 0$ so that for every sufficiently large even network size n and radius $r \leq r_c(n) = \sqrt{\frac{\alpha \log n}{n}}$, if $G \sim GK(n, r)$ and F is a random pairwise flow set, then with probability at least $1/2$ there exists $(s, \{x\}) \in F$ such that s is disconnected from x in G.*

At a high level, to prove this theorem we divide the unit square into a grid consisting of *boxes* of side length r, and then group these boxes into *regions* made up of 3×3 collections of boxes. If a given region has a node u in the center box, and all its other boxes are empty, then u is disconnected from any node not in its own box. Our proof calculates that for a sufficiently small constant fraction α used in the definition of the connectivity threshold, with probability at least $1/2$, there will be a node u such that u is isolated as described above, and u is part of a source/destination pair with another node v located in a different box.

Given this setup, the main technical complexity in the proof is carefully navigating the various probabilistic dependencies. One place where this occurs is in proving the likelihood of empty regions. For sufficiently small α values, the expected number of non-empty regions is non-zero, but we cannot directly concentrate on this expectation due to the dependencies between emptiness events. These dependencies, however, are dispatched by leveraging the negative association between the indicator variables describing a region's emptiness (e.g., if region i is not empty, this *increases* the chance that region $j \neq i$ is empty).

3.2.2 Bound on Achievable Throughput

In the previous section, we identified a radius threshold $r_c(n)$ below which a randomly generated network is likely to disconnect a source and destination, reducing the achievable throughput to a trivial 0. Here we study the properties of the networks generated with radius values on the other side of this threshold. In particular, we show that for any radius $r \geq r_c(n)$, with high probability, the randomly generated network and flow set will allow an optimal throughput bounded by $O(r)$. The intuition for this argument is that if nodes are evenly distributed in the unit square, a constant fraction of senders will have to deliver packets from one half of the square to the other, necessarily requiring many packets to flow through a small column in the center of the square, bounding the achievable throughput.

▶ **Theorem 3.7.** *For every sufficiently large even network size n and radius $r \geq r_c(n)$, given a network $G \sim GK(n,r)$ and a random pairwise flow set F, the throughput of every schedule (w.r.t. G and F) is $O(r)$ with high probability.*

3.2.3 Tightness of the Throughput Bound

In Section 3.2.2, we proved an upper bound of $O(r)$ on the achievable throughput in a network generated by $GK(n,r)$, for $r \geq r_c(n)$, and random pairwise flows. Here we show this result is tight by showing how to produce a schedule that achieves throughput in $\Omega(r)$ with respect to a random G and F. Formally:

▶ **Theorem 3.8.** *There exists a constant $\beta > 1$ such that, for any sufficiently large network size $n \geq 2$ and radius $r \geq \beta r_c(n)$, if $G \sim GK(n,r)$ and F is a random pairwise flow set, then with high probability in n there exists a schedule that achieves throughput in $\Omega(r)$ with respect to G and F.*

At a high level, our argument divides the unit square into box of side length $\approx r$. We prove that with high probability, both nodes and pairwise demands are evenly distributed among the boxes. This allows a schedule that efficiently moves many packets in parallel up and down columns to the row of their destination, and then moves these packets left and right along the rows to reach their destination. The time required for a given packet to make it to its destination is bounded by the column and row length of $\approx 1/r$, yielding an average throughput in $\Theta(r)$. The core technical complexity of this argument is the careful manner in which packets are moved onto and off a set of parallel paths while avoiding more than a small amount of congestion at any point in their routing.

4 Broadcast Capacity

The broadcast capacity problem assumes a designated *source* node has an infinite sequence of packets to spread to the entire network, implementing a one-to-all packet stream. Formally, this version of the capacity problem constrains the flow set to only contain a single pair of the form $\{s, V \setminus \{s\}\}$, for some source $s \in V$. As we will show, the achievable throughput for this problem in a given network graph G is strongly related to $d(G)$, the maximum degree of the minimum degree spanning tree (MDST) for G (see Section 2.3).

4.1 A Bound on Achievable Throughput for Arbitrary Networks

We establish that the maximum degree of an MDST in G – that is, $d(G)$ – bounds the achievable throughput, with larger values of $d(G)$ leading to lower throughput. The bound is primarily graph theoretic: arguing a fundamental limit on the rate at which packets can spread through a given topology.

▶ **Theorem 4.1.** *Fix a connected network graph $G = (V,E)$ and broadcast flow set F with source s. Then every schedule achieves throughput at most $O(1/d(G))$.*

Proof. Fix some $G = (V,E)$, $s \in V$, and $\mathcal{A}$, as specified by the theorem statement. If $d(G) \leq 4$ then the theorem is trivially true as all throughput values are in $O(1)$. Assume therefore that $d(G) > 4$. This allows us to apply Theorem 2.3 for $k = d(G) - 1$, which establishes that there exists a non-empty subset $S \subset V$ such that $c(G \setminus S) > q \cdot |S|$, for $q = k - 2 = d(G) - 3 > 1$ (where, as defined in Section 2.3, $c(G \setminus S)$ is the number of connected components after removing nodes in S from graph G).

Let C be the set of components in $G \setminus S$ that do not include the source s. Fix a packet t spread by s. We say t *arrives* at $C_i \in C$ in round $r \geq 1$, if this is the first round in which a node in C_i receives packet t. In this case, t must have been previously received by some bridge node in S that is adjacent to C_i. This holds because if t can make it from s's component to C_i without passing through a node in S, then removing S would not disconnect C_i.

Fix any packet count $i \geq 1$. Each packet requires $|C| = c(G \setminus S) - 1 \geq q|S|$ arrival events before it completes spreading. As we established above, each arrival event requires a given node in S to receive the given packet. Because each node in S can receive at most one packet per round, there are at most $|S|$ arrival events per round in the network.

Putting together these pieces, let T_i be the number of rounds required to spread i packets. We can lower bound this value as:

$$T_i \geq \frac{i \cdot |C|}{|S|} = \frac{i(q|S|)}{|S|} = iq \ .$$

It follows that for every schedule, and every i, at least T_i rounds are required to spread i packets – yielding a throughput upper bounded by $\frac{i}{T_i} \leq \frac{i}{i \cdot q} = 1/q = 1/(d(G) - 3)$, which yields the theorem. ◀

4.2 An Optimal Routing Algorithm for Arbitrary Networks

Here we describe a routing algorithm that achieves broadcast capacity throughput in $\Omega(1/d(G))$, when executed in a connected graph G. The high-level idea is to first construct an MDST T in the graph G. We then edge color T using $O(d(G))$ colors, and use this coloring to simulate the standard CONGEST model, parameterized so that a constant number of packets can fit within its bandwidth limit. We analyze a straightforward pipelining flooding algorithm for the CONGEST model that converges to constant throughput. When combined with our simulator, which requires $O(d(G))$ real rounds to simulate each CONGEST round, the result is a solution that achieves an average latency of $O(d(G))$ rounds per packet, providing the claimed $\Omega(1/d(G))$ throughput.

As in the pairwise setting, we can do this in a centralized fashion at the cost of a large convergence time (in particular, it takes up to $O(n^2)$ rounds to gather the graph topology locally before we can run a centralized algorithm). In order to decrease the convergence time, we describe in the full version a distributed version of this strategy that still converges to an optimal $\Omega(1/d(G))$ throughput in $O(n^2)$ rounds, but guarantees to converge to at least $\Omega\left(\frac{1}{d(G)+\log n}\right)$ throughput in $\tilde{O}(D(T) \cdot d(G) + \sqrt{n})$ rounds, where $D(T) \leq n$ is the diameter of a spanning tree T built by the algorithm and $\tilde{O}(\cdot)$ suppresses polylog(n) factors.

Formally, we prove the following theorem:

▶ **Theorem 4.2.** *There exists a (distributed) algorithm which, when executed in a connected network topology $G = (V, E)$ of size $n = |V|$, with a broadcast capacity flow set with source $s \in V$, achieves throughput in $\Omega(1/(d(G) + \log n))$ with convergence round $\tilde{O}(n \cdot d(G))$ and achieves throughput in $\Omega(1/d(G))$ with convergence round $O(n^2)$.*

4.3 Random Networks

The preceding broadcast capacity results hold for any connected network graph. Here we study the problem in networks randomly generated by the GK process with a communication radius sufficiently larger than the threshold $r_c(n)$.

Leveraging techniques from Section 3.2.3, we prove that such random networks are likely to contain a constant degree MDST, which, as established in Theorem 4.2, support constant throughput.

▶ **Theorem 4.3.** *There exists a (distributed) algorithm, such that for any sufficiently large network size $n > 1$ and constant $\beta \geq 1$, and radius $r \geq \beta r_c(n)$, if $G \sim GK(n,r)$ then with high probability the algorithm achieves constant throughput (for any s).*

5 All-to-All Capacity

We now consider the all-to-all capacity problem, which assumes all nodes begin with an infinite sequence of packets to spread to all other nodes. Formally, this variation of the capacity problem considers only the following canonical flow set: $F_{all} = \{(s, V \setminus \{s\}) : s \in V\}$.

In Section 4, we described and analyzed an algorithm that achieved a throughput in $\Omega(1/d(G))$ for delivering packets from a single source to the whole network. To solve all-to-all capacity, we could run n instances of this algorithm: one for each source, rotating through the different instances in a round robin fashion. This approach provides a baseline throughput result of $\Omega(1/(n \cdot d(G)))$. The key questions are whether or not this bound is tight, and whether there are simpler or more natural strategies than deploying round robin interleaving of single-source broadcast.

In the full version, we answer both questions in the affirmative by generalizing our argument from Theorem 4.1 to prove that no schedule achieves better than $O\left(\frac{1}{d(G)\cdot n}\right)$ throughput, and then exhibiting a matching distributed algorithm *SG* that uses a more natural strategy than round robin broadcast. Formally:

▶ **Theorem 5.1.** *When executed in a connected network topology $G = (V, E)$ of size $n = |V|$, with high probability in n: the* SG *algorithm achieves throughput in $\Omega\left(\frac{1}{d(G)\cdot n}\right)$ with respect to G and F_{all}. Furthermore, every schedule achieves throughput at most $O\left(\frac{1}{n\cdot d(G)}\right)$ with respect to G and F_{all}.*

Finally, notice that a direct corollary of our argument from Section 4.3, which establishes that a random graph contains a constant degree MDST (for sufficiently large radius) with high probability, is that with this same probability *SG* achieves $\Omega(1/n)$ throughput (which is best possible for all-to-all capacity).

5.1 Implications for One-Shot Gossip

Existing results for one-shot gossip in the mobile telephone model are expressed with respect to the vertex expansion (denoted α) of the graph topology [25, 24]. The best known results requires $O((n/\alpha)\text{polylog}(n))$ rounds, which is not tight in all graphs as vertex expansion does not necessarily characterize optimal gossip.[3] A key open question from [24] is whether it is possible to produce a gossip algorithm that is optimal (or within log factors of optimal) in *all* network topology graphs. The techniques used in the above capacity bounds help us prove the following, which largely resolves this open question:

[3] Consider, for example, a path of length n, which has $\alpha = 2/n$. It is possible to pipeline n messages through this network in $\Theta(n)$ rounds, which is much faster than $\tilde{O}(n/\alpha) = \tilde{O}(n^2)$.

▶ **Theorem 5.2.** *Fix a connected network topology $G = (V, E)$ with diameter D, size $n = |V|$, and MDST degree $d(G)$. Every solution to the one-shot gossip in G requires $\Omega(d(G) \cdot n)$ rounds. There exists an algorithm solves the problem in $O((D + \sqrt{n})\text{polylog}(n) + n(d(G) + \log n)) = \tilde{O}(d(G) \cdot n)$ rounds, with high probability in n.*

References

1. Gianluca Aloi, Marco Di Felice, Valeria Loscrì, Pasquale Pace, and Giuseppe Ruggeri. Spontaneous smartphone networks as a user-centric solution for the future internet. *IEEE Communications Magazine*, 52(12):26–33, 2014.
2. Apple. MultipeerConnectivity | Apple Developer Documentation. *Internet:* `https://developer.apple.com/documentation/multipeerconnectivity` *[Accessed : 07/20/2018]*, 2018.
3. Daniel Camps-Mur, Andres Garcia-Saavedra, and Pablo Serrano. Device-to-device communications with Wi-Fi Direct: Overview and experimentation. *IEEE Wireless Communications*, 20(3):96–104, 2013.
4. Flavio Chierichetti, Silvio Lattanzi, and Alessandro Panconesi. Rumour Spreading and Graph Conductance. In *Proceedings of the ACM-SIAM symposium on Discrete Algorithms (SODA)*, 2010.
5. Vasek Chvátal. Tough graphs and Hamiltonian circuits. *Discrete Mathematics*, 5(3):215–228, 1973.
6. Sebastian Daum, Fabian Kuhn, and Yannic Maus. Rumor spreading with bounded in-degree. In *Proceedings of the International Colloquium on Structural Information and Communication Complexity*, 2016.
7. Michael Dinitz, Jeremy Fineman, Seth Gilbert, and Calvin Newport. Load balancing with bounded convergence in dynamic networks. In *INFOCOM*, pages 1–9, 2017.
8. Michael Dinitz, Magnús M. Halldórsson, Taisuke Izumi, and Calvin Newport. Distributed Minimum Degree Spanning Trees. In *Proceedings of the ACM Symposium on the Principles of Distributed Computing (PODC)*, 2019 (to appear.).
9. Nikolaos Fountoulakis and Konstantinos Panagiotou. Rumor spreading on random regular graphs and expanders. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 560–573. Springer, 2010.
10. Alan M Frieze and Geoffrey R Grimmett. The shortest-path problem for graphs with random arc-lengths. *Discrete Applied Mathematics*, 10(1):57–77, 1985.
11. Martin Fürer and Balaji Raghavachari. Approximating the Minimum-Degree Steiner Tree to within One of Optimal. *Journal of Algorithms*, 17(3):409–423, 1994.
12. Open Garden. FireChat. *Internet:* `https://www.opengarden.com/` *[Accessed: 07/20/2018]*, 2018.
13. Open Garden. The Open Garden Hotspot. *Internet:* `https://www.opengarden.com/` *[Accessed: 07/20/2018]*, 2018.
14. Mohsen Ghaffari and Calvin Newport. How to Discreetly Spread a Rumor in a Crowd. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, 2016.
15. George Giakkoupis. Tight bounds for rumor spreading in graphs of a given conductance. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science (STACS)*, 2011.
16. George Giakkoupis. Tight bounds for rumor spreading with vertex expansion. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2014.
17. George Giakkoupis and Thomas Sauerwald. Rumor spreading and vertex expansion. In *Proceedings of the ACM-SIAM symposium on Discrete Algorithms (SODA)*, pages 1623–1641. SIAM, 2012.
18. Carles Gomez, Joaquim Oller, and Josep Paradells. Overview and evaluation of Bluetooth low energy: An emerging low-power wireless technology. *Sensors*, 12(9):11734–11753, 2012.

19 Piyush Gupta and Panganmala R Kumar. The capacity of wireless networks. *IEEE Transactions on Information Theory*, 46(2):388–404, 2000.

20 Adrian Holzer, Sven Reber, Jonny Quarta, Jorge Mazuze, and Denis Gillet. Padoc: Enabling social networking in proximity. *Computer Networks*, 111:82–92, 2016.

21 Zongqing Lu, Guohong Cao, and Thomas La Porta. Networking smartphones for disaster recovery. In *Proceedings of the IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 1–9. IEEE, 2016.

22 Aleksander Madry. Faster approximation schemes for fractional multicommodity flow problems via dynamic graph algorithms. In *Proceedings of the Forty-Second ACM Symposium on Theory of Computing (STOC)*, pages 121–130. ACM, 2010.

23 David Mark, Jayant Varma, Jeff LaMarche, Alex Horovitz, and Kevin Kim. Peer-to-Peer Using Multipeer Connectivity. In *More iPhone Development with Swift*, pages 239–280. Springer, 2015.

24 Calvin Newport. Leader Election in a Smartphone Peer-to-Peer Network. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017. Full version available online at: `http://people.cs.georgetown.edu/~cnewport/pubs/le-IPDPS2017.pdf`.

25 Calvin Newport and Alex Weaver. Random Gossip Processes in Smartphone Peer-to-Peer Networks. In *Proceedings of the International Conference on Distributed Computing in Sensor Systems (DCOSS)*, 2019.

26 DG Reina, Mohamed Askalani, SL Toral, Federico Barrero, Eleana Asimakopoulou, and Nik Bessis. A survey on multihop ad hoc networks for disaster response scenarios. *International Journal of Distributed Sensor Networks*, 11(10):647037, 2015.

27 Claude E Shannon. A theorem on coloring the lines of a network. *Journal of Mathematics and Physics*, 28(1-4):148–152, 1949.

28 Noriyuki Suzuki, Jane Louie Fresco Zamora, Shigeru Kashihara, and Suguru Yamaguchi. Soscast: Location estimation of immobilized persons through sos message propagation. In *Proceedings of the International Conference on Intelligent Networking and Collaborative Systems (INCoS)*, pages 428–435. IEEE, 2012.

29 Sein Win. On a connection between the existence of k-trees and the toughness of a graph. *Graphs and Combinatorics*, 5(1):201–205, 1989.

Sublinear-Time Distributed Algorithms for Detecting Small Cliques and Even Cycles

Talya Eden
Electrical Engineering Department, Tel-Aviv University, Israel
talyaa01@gmail.com

Nimrod Fiat
Computer Science Department, Tel-Aviv University, Israel
nimrodf1@tau.ac.il

Orr Fischer
Computer Science Department, Tel-Aviv University, Israel
orrfischer@mail.tau.ac.il

Fabian Kuhn
Computer Science Department, University of Freiburg, Germany
kuhn@cs.uni-freiburg.de

Rotem Oshman
Computer Science Department, Tel-Aviv University, Israel
roshman@mail.tau.ac.il

Abstract

In this paper we give sublinear-time distributed algorithms in the CONGEST model for subgraph detection for two classes of graphs: cliques and even-length cycles. We show for the first time that all copies of 4-cliques and 5-cliques in the network graph can be listed in sublinear time, $O(n^{5/6+o(1)})$ rounds and $O(n^{21/22+o(1)})$ rounds, respectively. Prior to our work, it was not known whether it was possible to even check if the network *contains* a 4-clique or a 5-clique in sublinear time.

For even-length cycles, C_{2k}, we give an improved sublinear-time algorithm, which exploits a new connection to extremal combinatorics. For example, for 6-cycles we improve the running time from $\tilde{O}(n^{5/6})$ to $\tilde{O}(n^{3/4})$ rounds. We also show two obstacles on proving lower bounds for C_{2k}-freeness: First, we use the new connection to extremal combinatorics to show that the current lower bound of $\tilde{\Omega}(\sqrt{n})$ rounds for 6-cycle freeness cannot be improved using partition-based reductions from 2-party communication complexity, the technique by which all known lower bounds on subgraph detection have been proven to date. Second, we show that there is some fixed constant $\delta \in (0, 1/2)$ such that *for any* k, a $\Omega(n^{1/2+\delta})$ lower bound on C_{2k}-freeness implies new lower bounds in circuit complexity.

For general subgraphs, it was shown in [14] that for any fixed k, there exists a subgraph H of size k such that H-freeness requires $\tilde{\Omega}(n^{2-\Theta(1/k)})$ rounds. It was left as an open problem whether this is tight, or whether some constant-sized subgraph requires *truly quadratic* time to detect. We show that in fact, for any subgraph H of constant size k, the H-freeness problem can be solved in $O(n^{2-\Theta(1/k)})$ rounds, nearly matching the lower bound of [14].

2012 ACM Subject Classification Networks → Network algorithms; Theory of computation → Distributed algorithms; Theory of computation → Lower bounds and information complexity

Keywords and phrases Distributed Computing, Subgraph Freeness, CONGEST

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.15

Related Version A full version of the paper is available at `https://www.cs.tau.ac.il/~roshman/papers/DISC19_EFFKO19.pdf`.

Funding *Talya Eden*: The Israel Science Foundation grant No.1146/18. Talya Eden is greatful to the Azrieli Foundation for an award of an Azrieli Fellowship.
Nimrod Fiat: The Israel Science Foundation grant No.1146/18 and the Kadar family award.
Orr Fischer: The Israeli Centers of Research Excellence (I-CORE) program (Center No. 4/11) and the Laura Schwarz-Kipp Institute of Computer Networks.
Rotem Oshman: The Israeli Centers of Research Excellence (I-CORE) program (Center No. 4/11).

© Talya Eden, Nimrod Fiat, Orr Fischer, Fabian Kuhn, and Rotem Oshman;
licensed under Creative Commons License CC-BY

33rd International Symposium on Distributed Computing (DISC 2019).
Editor: Jukka Suomela; Article No. 15; pp. 15:1–15:16

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

In the subgraph-freeness problem, a network must decide whether its communication graph contains a copy of some fixed subgraph H or not. If the network is H-free, then all nodes should accept, but if the graph contains a copy of H, then at least one node should reject. The subgraph-freeness problem has received significant attention in the sequential world, and recently also in the distributed community. Other than being a fundamental graph problem, it has many application in other scientific fields such as biology and social sciences (e.g [27, 28, 25]).

From the theoretical perspective, distributed subgraph freeness is especially interesting because it is an extremely *local* problem: to solve H-freeness for a graph H of size k, each node only needs to examine its own k-hop neighborhood. However, it is known that in bandwidth-constrained networks (the CONGEST model), subgraph freeness cannot always be solved efficiently [11, 10, 23, 19, 21, 14]. In fact, it is not even known which classes of subgraphs H admit a *sublinear-round* distributed algorithm for H-freeness: some simple subgraphs, like odd-length cycles, are known to require linear time [11], and some subgraphs even require nearly-quadratic time [14]. In contrast, for triangles [5] and for even-length cycles [14], sublinear-time algorithms are known.

We seek to improve our understanding of sublinear-time algorithms for two classes of subgraphs: *cliques* and *even-length cycles.* We also show that any constant-sized subgraph can be detected in sub-quadratic time.

Small cliques. We show for the first time that 4-cliques and 5-cliques can be detected in sublinear time; previously, no non-trivial algorithm for K_4-freeness or listing was known, and the same is true for K_5 (the trivial solution is simply to have each node send its entire neighborhood to all its neighbors, which requires $\Theta(n)$ rounds). In fact, our algorithm is even able to *list* all copies of K_4, K_5:

▶ **Theorem 1.** *The problem of enumerating all 4-cliques in the graph can be solved in $\tilde{O}(n^{5/6+o(1)})$ rounds in* CONGEST*, and all 5-cliques can be enumerated in $\tilde{O}(n^{21/22+o(1)})$ rounds.*

Our algorithm builds on a recent approach of [5], which decomposes the graph into well-connected clusters and a sparse set of edges, and uses this to give an elegant algorithm that enumerates all triangles. A triangle that has two or three edges in the sparse part of the graph can be found using the sparsity of this edge set, while a triangle that has two edges in a well-connected cluster can be found by the cluster nodes. Unlike triangles, however, a 4-clique could be "split" between two clusters, with one edge in one cluster, another edge in another cluster, and the remaining edges crossing between the two clusters in the sparse part of the graph. With a 5-clique the situation becomes even more complex. Thus, listing all 4-cliques and 5-cliques requires significant effort and new ideas beyond [5].

Even-length cycles. Turning our attention to even-length cycles, C_{2k} for constant k, we give an improved sublinear-time algorithm for C_{2k}-freeness (it is known that odd-length cycles require linear time [11]). Our improved algorithm exploits a new connection to extremal combinatorics: we show that the *Zarankiewitz number* of the cycle C_{2k}, which is the maximum number of edges in a *bipartite* graph that does not contain C_{2k}, plays a role in testing C_{2k}-freeness, even for non-bipartite graphs. This allows us to modify the algorithm from [14] and improve its running time:

▶ **Theorem 2.** *C_{2k}-freeness can be solved in at most $\tilde{O}_k(n^{1-2/(k^2-k+2)})$ rounds for odd $k \geq 3$, and at most $\tilde{O}_k(n^{1-2/(k^2-2k+4)})$ rounds for even $k \geq 4$.*

For example, for 6-cycles we improve the running time from $\tilde{O}(n^{5/6})$ (in [14]) to $\tilde{O}(n^{3/4})$ rounds.

We remark that while our K_4 and K_5 algorithms list all copies of K_4, K_5 in the graph, our algorithm for even-length cycles is only able to *check* if the graph contains a copy of C_4. In general, cliques seem like a more "local" type of subgraph than cycles: the presence of a clique implies that all its nodes can communicate with each other, which is obviously not true for cycles. We formalize this intuition by showing that short cycles really are different from small cliques – it is not possible to enumerate all of them in sublinear time:

▶ **Theorem 3.** *Enumerating all C_4 in the graph requires $\tilde{\Theta}(n)$ rounds.*

Obstacles on proving lower bounds for even-length cycles. For any $k \geq 2$, the lower bound for C_{2k}-freeness has been "stuck" at $\Omega(\sqrt{n})$ for a long time [11, 21]. We give two reasons for why improving this lower bound might be hard.

First, using the same connection to Zarankiewicz numbers, we show that reductions to two-party communication complexity, of the type used to prove all known lower bounds on H-freeness for any subgraph H [11, 14, 21, 8], cannot be used to give a lower bound better than $\tilde{\Omega}(\sqrt{n})$ on C_6-freeness. Following [8], which showed a similar result for cliques, we show:

▶ **Theorem 4.** *Let (V_A, V_B) be a partition of the vertices V of the graph between two players, and assume that each player initially knows the edges adjacent to any node on its side of the graph (V_A or V_B, respectively). If the cut (V_A, V_B) contains s edges, then there is a two-party protocol that communicates $\tilde{O}(\sqrt{n} \cdot s)$ bits and solves C_6-freeness.*

Our protocol uses different ideas from the two-party protocol of [8]; to bound the number of edges the players need to send each other, we rely on results from extremal combinatorics.

Next, we show that lower bounds on C_{2k} are related to circuit lower bounds, in the following sense:

▶ **Theorem 5** (Informal). *There exists an absolute constant $c < 1$ such that for any $k \geq 3$, proving a lower bound of $\Omega(n^{1-c})$ on C_{2k}-hardness would imply new lower bounds on high-depth circuits with constant fan-in and fan-out.*

We show this by extending a connection shown in [11] between lower bounds for the Congested Clique and circuit complexity to high-conductance components.

General subgraphs. It was shown in [14] that some subgraphs are very hard to detect: for any $k \geq 4$, there exists a graph on k vertices that requires $\tilde{\Omega}(n^{2-\Theta(1/k)})$ rounds to detect. It was left open whether this bound is tight, or whether the loss of $1/k$ in the exponent is an artifact of the proof: the lower bound of [14] is shown by a reduction from two-party communication complexity, where the graph is partitioned into two parts, with a cut of size $\Theta(n^{1/k})$ between them. This causes the lower bound to "lose" a factor of $n^{1/k}$.

We show that, surprisingly, the factor $n^{1/k}$ is not just an artifact of the proof:

▶ **Theorem 6.** *For any constant k and subgraph H of size k, the H-freeness problem can be solved in $O(n^{2-2/(3k+1)+o(1)})$ rounds in* CONGEST*.*

Thus, subgraph-freeness is not "maximally hard" in CONGEST: it does not require truly quadratic time.

1.1 Related Work

The problems of subgraph-freeness and listing have been extensively studied in both the centralized and the distributed settings; for lack of space, we mention here the most directly related results. In [5, 6] randomized algorithms based on expander decompositions for listing all triangles were shown, culminating in an $\tilde{O}(n^{1/3})$ round algorithm in the CONGEST model. This improved the previous algorithm of [20]. The algorithms of [5, 6] finds a *conductance decomposition* of the graph, and then uses a routing result of [17, 18] to quickly find all triangles contained or adjacent to some cluster in the decomposition. Our K_4-listing algorithm uses the decomposition from [5], albeit in a somewhat different manner from the way it is used in [5]. We also use the K_s-listing algorithm for the Congested Clique of [10] as a subroutine; [10] shows that all copies of K_s can be found in $O(n^{1-2/s})$ in the Congested Clique. To our knowledge, prior to our work, no sublinear-time K_s-freeness algorithm was known for any $s \geq 4$. However, in [1], it is shown that if the network contains an ϵ-*near-clique* of linear size, then an ϵ^3-near clique of linear size can be found in constant time. For 4-cycles, C_4, a nearly-tight bound of $\tilde{\Theta}(\sqrt{n})$ was shown in [11]. The lower bound was extended to an $\tilde{\Omega}(\sqrt{n})$ lower bound for any even-length cycle C_{2k} in [21]. In the Congested Clique, an $O(n^{0.158})$ round algorithm for C_k-detection was shown by [4] based on algebraic methods. The first sublinear-time algorithm for C_{2k}-freeness for $k \geq 3$ was given in [14], and we improve this algorithm here. It is known that odd-length cycles require nearly-linear time to detect [11]. It is shown in [14] that some subgraphs of size $O(k)$ require $\tilde{\Omega}(n^{2-\frac{1}{k}})$ rounds to detect (for any constant $k \geq 4$). There are algorithms for clique-freeness and cycle-freeness in related models, e.g., [4, 10, 3, 16, 13, 15], but they are not directly relevant to our work, except as mentioned above.

2 Preliminaries

The CONGEST model. The CONGEST model is a synchronous network model, where computation proceeds in *rounds*. The network is modeled as a graph, $G = (V, E)$. Each graph node $v \in V$ initially knows its own neighborhood, denoted $N(v)$. It is assumed that nodes have unique identifiers, which we conflate with V. In each round, each node of the network may send $O(\log n)$ bits on each of its edges, and these messages are received by neighbors in the current round.

Some of our algorithms rely on results from the *Congested Clique* model. As in CONGEST, we have an input graph $G = (V, E)$, where each vertex V is a separate computing node, which initially knows its neighborhood. However, unlike CONGEST, in the Congested Clique, all the nodes can talk directly to each other: in each round, each node can send $O(\log n)$ bits to every other node in V, even if the edge between them is not in E. The Congested Clique admits very efficient algorithms for many distributed tasks, and our algorithms build on a clique detection algorithm for this model [10] by *simulating* it in regular CONGEST.

The main problem we are concerned with in this paper is the following:

▶ **Definition 7** (Subgraph freeness and enumeration). *Fix a constant-sized graph H.*

*In the H-*freeness *problem, the goal is to determine whether the input graph G contains a copy of H as a subgraph or not. If G is H-free, then all nodes should accept, but if G contains H as a subgraph, then at least one node should reject.*

*In the H-*enumeration (or listing) *problem, each node outputs a (possibly empty) set of copies of H in G, such that together the nodes output all copies of H in G.*

For a graph G, we let $V(G)$ denote the vertex set of G, and $E(G)$ denote the edges of G.

The *arboricity* of the graph is defined as the minimum number of edge-disjoint forests required to cover the graph edges. A graph with m edges has arboricity at most $O(\sqrt{m})$ [7].

3 Enumerating All 4-Cliques in Sublinear Time

In this section we show how to find all copies of K_4 in the network graph in $O(n^{5/6+o(1)})$ rounds. Throughout the section, we will use two parameters: $\epsilon = 1/2, \delta = 5/6$. Since some parts of the algorithm will be re-used in later sections with different values for ϵ, δ, we leave our results parameterized.

On a very high level, the algorithm works as follows: first, we decompose the edges E of the graph into two sets, E_s and E_m, by recursively applying a graph decomposition from [5]. The set E_s has the property that every node v has at most n^δ edges in E_s; therefore, cliques contained entirely in E_s are easy to find, by simply having all nodes announce to all their neighbors all their edges in E_s.

The set E_m is the edge-disjoint union of $O(n^{1-\delta})$ "very well-connected" clusters of nodes. On each such cluster, we can fairly efficiently simulate the Congested Clique algorithm for finding 4-cliques of [10]. However, we need to find *all* 4-cliques that have at least one edge in E_m. This includes cliques $\{u_0, u_1, u_2, u_3\}$ such that $\{u_0, u_1\} \in E_m$, so that nodes u_0, u_1 are together in some cluster C, but nodes u_2, u_3 are outside cluster C; while nodes u_0, u_1 together know about almost all edges of the clique, the edge $\{u_2, u_3\}$ is not necessarily known to any node of C. If we want to find such cliques by simulating a Congested Clique algorithm on C, we must first "bring in" edges from outside C, so that the cluster nodes know about them and can use them in the simulation.

There can be $\Theta(n^2)$ edges outside of C, and we cannot afford to have *all* of them sent to nodes of C, because this would require too much time. We resolve this difficulty by considering two types of "external nodes". If u_2, u_3 do not have many neighbors in C, then we can use this fact to efficiently find all cliques containing u_2, u_3 and two nodes from C. On the other hand, if either u_2 or u_3 does have many neighbors in C, they can quickly send their entire neighborhood to the nodes of C, by splitting their neighborhood and sending each part of it on a different edge to C. Then, the cluster nodes, having learned about the external edges, will use them in their simulation of the Congested Clique, and find any 4-clique containing u_2, u_3.

We now describe our algorithm in more detail. We begin by showing how we apply the conductance decomposition from [5] in a slightly different manner than [5] uses it, and then how we find 4-cliques on the resulting decomposed graph.

3.1 Conductance Decomposition

A main ingredient in the algorithm is the conductance decomposition developed in [5], which partitions the edges E of the graph into three sets, E_m, E_s, E_r, such that E_m induces a set of "well-connected clusters", E_s induces a low-degree graph, and $|E_r| \leq |E|/6$. The well-connected clusters in E_m each satisfy the following:

▶ **Definition 8** (n^δ-cluster). *A subset $G' = (V', E')$ of G is called an n^δ-*cluster *(or simply "cluster" for short) if it satisfies the following conditions:*
1. *It has $O(\text{polylog}(n))$ mixing time,*
2. *End each vertex $v \in V'$ has degree $\Omega(n^\delta)$ in E'.*

The *mixing time* of a graph is the time required for a random walk on the graph to approach its stationary distribution; the precise definition is not needed for our purposes here, because as in [5], we use the mixing-time guarantee to apply a routing result from [17] as a black box.

In [5], the algorithm finds triangles by (a) decomposing the edge-set, (b) finding triangles that include an edge in E_s or E_m, and then (c) recursing on E_r. (Edges are sometimes moved from E_m to E_r, but not too many.) In contrast, for our purposes here, we must first eliminate all the edges of E_r: we have no guarantee on these edges other than the fact that there are not too many of them, and to find a 4-clique touching E_m or E_s that may include edges from E_r, this is not sufficient. By recursively applying the decomposition from [5], we obtain the following decomposition of the graph.

▶ **Lemma 9.** *Fix a graph $G = (V, E)$ with $|V| = n$ and diameter D. We can find, w.h.p., in $\tilde{O}(n^{1-\delta} + D)$ rounds, a decomposition of E to $E = E_m \cup E_s$ satisfying the following conditions:*

(a) *E_m is the union of at most $s = O(\log n)$ sets, $E_m = \bigcup_{i=1}^{s} E_m^i$, where each E_m^i is the vertex-disjoint union of $O(n^{1-\delta})$ n^δ-clusters, $C_i^1, \ldots, C_i^{k_i}$.*
The set E_m^i is called the i-th level *of the decomposition. We say that a node u* belongs *to cluster C_i^j if at least one of u's edges is in C_i^j.*

(b) *Each level-i cluster C_i^j has a unique identifier, which is a pair of the form (i, x) where $x \in [n^{1-\delta}]$, and a unique leader node, which is some node in the cluster.*
Each node u knows the identifiers of all the clusters C_i^j to which u belongs, the leaders for those clusters, and it knows which of its edges belong to which clusters.

(c) *$E_s = \bigcup_{v \in V} E_{s,v}$, where $E_{s,v}$ is a subset of edges incident to v and $|E_{s,v}| \leq n^\delta \log n$. Each vertex v knows $E_{s,v}$.*

To assign the clusters unique identifiers and leaders, we rely on our "diameter reduction" technique, which allows us to assume w.l.o.g. that the diameter of the network is $D = O(\text{polylog}(n))$ (See full version for details [12]). Thus, in $\tilde{O}(n^{1-\delta})$ rounds, we can select the smallest node in each cluster, and disseminate the IDs of these nodes throughout the network.

In the sequel we abuse notation slightly, by thinking of a cluster C_i^j as both a set of edges and the set of nodes that belong to the cluster.

3.2 Finding 4-Cliques

In the remainder of the section we describe our 4-clique listing algorithm; the runtime analysis and correctness proof appear in the full version of the paper [12].

We begin by computing the decomposition from Lemma 9. Next, we look for cliques entirely contained in E_s, the "low-degree" part of the graph. Then we turn to the harder task, finding cliques that include at least one edge of E_m (i.e., a cluster edge). As we explained above, we divide these cliques into two types: those that have two nodes external to the cluster with few neighbors in the cluster, and those that do not.

Step 1: Finding cliques contained in E_s. To find 4-cliques contained entirely in E_s, we simply have each node v send all of $E_{s,v}$ to all its neighbors. As $|E_{s,v}| \leq n^\delta$, this can be done in n^δ rounds. Any node that sees a copy of K_4 outputs it. It is easy to verify that any 4-clique whose edges are all in E_s will be detected by all four of its vertices.

Step 2: Finding cliques containing an edge from E_m. Next, we search for copies of K_4 that have at least one edge in some cluster. We divide these cliques into two types.

Let $\epsilon \in (0,1)$ be a parameter (as we said above, in this section we will use $\epsilon = 1/2$, but the next section re-uses some of the machinery we develop here with a different ϵ). Given a cluster C, we say that a node v is C*-light* if $v \notin C$ and v has at most n^ϵ neighbors in C (that is, $|N(v) \cap C| \le n^\epsilon$). If v is not C-light, then we say that v is C*-heavy*.

Now let H be a copy of K_4 that has at least one edge in E_m (that is, in some cluster). We say that H is *light* if H contains at least two nodes that are C-light for some cluster C. Otherwise, we say that H is *heavy*.

Step 2(a): Finding light cliques. To find cliques containing at least two nodes that are light with respect to some cluster, we iterate through the clusters sequentially, in lexicographic order of their cluster IDs.

For each C, all nodes that belong to C (i.e., have at least one edge in C) announce this fact to their neighbors. Let $M_C(v) = N(v) \cap C$ denote the neighbors of vertex v that belong to C.

Next, each node v that is C-light sends $M_C(v)$ to all its neighbors; this requires $O(n^\epsilon)$ rounds, as v is C-light. Upon receiving $M_C(u)$ from each C-light neighbor $u \in N(u)$, node v forms a list of "candidates" – triplets of nodes that may complete a K_4 with v:

$$\mathcal{L}_v = \{\{u, c_1, c_2\} \mid u \in N(v) \text{ is } C\text{-light, } c_1, c_2 \in C, \text{ and } c_1, c_2 \in M_C(u) \cap M_C(v)\}.$$

For each candidate $\{u, c_1, c_2\}$, node v already knows that all edges of the 4-clique on $\{v, u, c_1, c_2\}$ are present, except for edge $\{c_1, c_2\}$, which may or may not be present. To check, node v goes through its candidates, and sends each cluster neighbor $c_1 \in C$ a list of "edge queries",

$$\mathcal{Q}_{v,c_1} = \{\{c_1, c_2\} \mid \exists u \in N(v) : \{u, c_1, c_2\} \in \mathcal{L}_v\},$$

the list of potential edges of c_1 that, if present, would complete a 4-clique. There are at most n^ϵ such edges: by definition of $\mathcal{L}_v$, if $\{c_1, c_2\}$ is sent to c_1, then $c_2 \in M_C(v)$, but $|M_C(v)| \le n^\epsilon$ (as v is C-light). Node c_1 responds with the subset $(\{c_1\} \times N(c_1)) \cap \mathcal{Q}_{v,c_1}$ of edges that are actually present, and node v outputs the 4-cliques it has found.

Step 2(b): Finding heavy cliques. Finally, we look for 4-cliques where at least one edge is in a cluster, and the other two nodes are not light w.r.t. that cluster; they might be *in* the cluster, or they might be outside it but have many neighbors in the cluster.

Let $F(C) = \{\{u, v\} \mid u \in C \text{ or } u \text{ is } C\text{-heavy}\}$ be the set of all edges adjacent to C or to some C-heavy node. We iterate through the levels $i = 1, \ldots, s$ of the decomposition; our goal is to find 4-cliques contained entirely in $F(C_i^j)$, in parallel for all the step-i clusters, $C_i^1, \ldots, C_i^{k_i}$. To do this, we have the cluster nodes simulate the execution of the Congested Clique algorithm for K_4 enumeration from [10] on the n-vertex graph $(V, F(C))$. This part of the algorithm is carried out in four steps:

I. **Pull:** the nodes of the cluster C "pull" the edges of $F(C)$ from nodes outside C, such that every edge of $F(C)$ is learned by at least one node in C.

II. **Partition:** in this step, we compute a partition $\{V_c\}_{c \in C}$ of V, which is roughly balanced (i.e., $|V_{c_1}| \approx |V_{c_2}|$ for each $c_1, c_2 \in C$). Each node $c \in C$ will be responsible for simulating the nodes in V_c.

III. Shuffle: the cluster nodes shuffle the edges of $F(C)$ between themselves, so that each node $c \in C$ learns the $F(C)$-neighborhood $N_{F(C)}(v)$ for each node $v \in V_c$ it needs to simulate. This is carried out by using the routing algorithm from [17].

IV. Simulate: the nodes of C simulate an n-vertex Congested Clique, and run the K_4-enumeration algorithm of [10] on the graph $(V, F(C))$ to list all the 4-cliques in $F(C)$.

▶ Remark 10. It is worth noting that [5] gives an extension of the Congested Clique algorithm of [10] to any graph with a low mixing time. This is unfortunately unsuited for our purposes, because in our case we do not run the algorithm only on edges adjacent to cluster nodes, but instead on a potentially much larger set, $F(C)$. In addition, we simulate an Congested Clique of size n on a n^δ-cluster, while in [5] the clusters do not need to simulate any external nodes.

As we said, these four steps are carried out *in parallel* for all the level-i clusters. Other than the first step, the remaining steps involve only intra-cluster communication (i.e., communication between nodes of the same cluster over the edges belonging to the cluster). Because the level-i clusters are vertex-disjoint components, we incur no extra congestion from simulating all level-i clusters in parallel.

We elaborate on each step below.

Pull. For each level-i cluster C in parallel, the nodes do the following. For a node v, let $S(v) = \{\{v, u\} \mid u \in N(v) \wedge v < u\}$ be the edges adjacent to v in which v has the smaller ID.

Each C-heavy node v that is not in C partitions its edges $S(v)$ into $|M_C(v)|$ sets, each of size at most $|S(v)|/|M_C(v)| \leq n/n^\epsilon = n^{1-\epsilon}$, and sends each set to a different neighbor in C, in $O(n^{1-\epsilon})$ rounds.

Following this step, each edge in $F(C)$ is known to exactly one node in C; we define the *initial load* of node $c \in C$ to be the number of $F(C)$-edges it knows (including its own edges).

▶ **Observation 11.** *Since each C-heavy node partitions its edges into at least n^ϵ sets, and $c \in C$ has at most n neighbors that are C-heavy, the initial load of c is at most $n \cdot n^{1-\epsilon} = n^{2-\epsilon}$.*

Partition. The nodes of C must now partition all the graph nodes V among themselves, in a roughly-balanced way, quickly and without a lot of communication. Every cluster node needs to learn the *entire* partition, not just its own part, so that it knows which node of V will be simulated by whom.

Ideally, we would assign each node of V to a uniformly random node in C, but this would require a lot of communication. Instead, we use a family of $O(\log n)$-wise independent hash functions to ensure a relatively balanced partition, allowing us to represent the entire partition using only $O(\log^2 n)$ bits.

For convenience, we use only n^δ nodes, which we call the *active nodes*, to carry out the simulation; the cluster leader selects these nodes by choosing n^δ of its neighbors, $u_1, \ldots, u_{n^\delta}$. The leader also selects an ℓ-wise independent hash function $f : U \to [n^\delta]$, where $\ell = O(\log n)$, and U is the domain from which IDs for the graph are drawn. Then the leader disseminates the assignment of active nodes, $\{(i, u_i)\}_{i=1,\ldots,n^\delta}$, and the $O(\log^2 n)$-bit representation of f to all the cluster nodes. Using pipelining, this requires $\tilde{O}(n^\delta)$ rounds. (We note that since the cluster has polylogarithmic mixing time, it also has polylogarithmic diameter.)

Now let $V_{u_i} = \{v \mid f(v) = i\}$ be the set of nodes that active node u_i will simulate. Using a concentration result from [24], for $O(\log n)$-wise independent random variables, we obtain:

▶ **Lemma 12.** *Let $\ell = 4 \log n$, and assume that f is ℓ-wise independent. Then with probability at least $1 - 1/n$, we have $|V_{u_i}| \leq 2n^{1-\delta}$ for each active node u_i.*

Shuffle and Simulate. Next, we must route the edges of $F(C)$, from the cluster nodes that know them initially to the nodes that need them for the simulation. Then we run a simulation of the algorithm of [10], where each node in C simulates at most $O(n^{1-\delta})$ nodes of G. For both purposes we use the routing scheme of [17], which allows us to deliver roughly n^{δ} messages from each cluster node to other cluster nodes in $n^{o(1)}$ rounds.

There are some technical details involved in the simulation, because we must ensure that nodes do not try to send or receive too many messages at once. These details are deferred to the full version of the paper. Ultimately, we show the following result:

▶ **Lemma 13.** *For a constant $0 < \epsilon \leq 1$, suppose an edge set E' is partitioned between the nodes of C, so that each node $u \in C$ initially knows a subset E'_u of size at most $O(n^{2-\epsilon})$. Then a simulation of t rounds of the Congested Clique on $G' = (V, E')$ can be performed in $O((n^{2-\delta-\epsilon} + t \cdot n^{2-2\delta}) \cdot n^{o(1)})$ rounds, with success probability $1 - \frac{1}{n^2}$.*

Using this simulation, each cluster simulates the K_4 enumeration algorithm of [10]. In the full Congested Clique, this algorithm runs in $O(\sqrt{n})$ rounds, but since we are using clusters that simulate one round of the Congested Clique in roughly $n^{2-2\delta}$ rounds, our running time will be $O(n^{2-2\delta+1/2})$. This completes our high-level overview of the K_4-enumeration algorithm.

3.3 Listing 5-Cliques

We outline at a high level the changes needed to list all 5-cliques.

The main difficulty in moving from 4-cliques to 5-cliques is that a 5-clique can be partitioned between clusters, so that **(a)** each edge of the 5-clique belongs to a different cluster (recall that the clusters constructed in Lemma 9 are not vertex-disjoint), and furthermore, **(b)** no cluster is able to "pull in" all the edges of the 5-clique, because from the perspective of each cluster, of the three nodes outside the cluster, two are light and one is heavy.

We modify our framework so that it can handle 5-cliques by making two main changes:

(1) We do not call the decomposition from [5] recursively. Instead, we execute only one step of the decomposition, which yields a partition of the edges E into a set of well-connected, vertex-disjoint clusters; a sparse set, E_s; and the "remainder", E_r, which is of size at most $|E_r| \leq \alpha \cdot |E|$.

(2) We change the parameters of the decomposition so that E_r, the "remaining edges", constitute a *sub-constant* fraction of the total number of edges (i.e., $\alpha = o(1)$), see [9]. (We pay for this by an increased mixing time.) This implies that the arboricity of E_r is *sublinear*, as any graph with m edges has arboricity at most $\sqrt{m}$.

We are then able to overcome the difficulties and find all copies of K_5 using the fact that the edges outside the clusters have sublinear arboricity, and that the clusters are vertex-disjoint; the resulting running time is $O(n^{21/22+o(1)})$ rounds.

4 Improved Algorithm for C_{2k}-Freeness

We now turn out attention to even-length cycles, and give an improved algorithm for C_{2k}-freeness.

Our main technical contribution is to show that, because of a new connection to Zarankiewicz numbers, if we have a graph that is C_{2k}-free, then this graph cannot have too many high-degree nodes. The bound we obtain is tighter than a bound used in [14], and it yields an improved algorithm for C_{2k}-freeness.

▶ **Definition 14** (Zarankiewicz numbers [26]). *The Zarankiewicz number $z(a, b, H)$ is defined as the maximum number of edges in an (a, b)-bipartite graph that does not contain H as a subgraph.*

We rely on the following upper bound:

▶ **Theorem 15** (Zarankiewicz numbers for cycles [26, 22]). *For any integers $a, b \geq 0$ and $k \geq 2$,*

$$z(a, b, C_{2k}) \leq \begin{cases} (2k-3)\left((ab)^{\frac{1}{2}+\frac{1}{2k}} + a + b\right), & \text{for odd } k, \\ (2k-3)\left(a^{\frac{k+2}{2k}} b^{\frac{1}{2}} + a + b\right), & \text{for even } k. \end{cases}$$

Using Theorem 15, we can better bound the number of high-degree vertices in C_{2k}-free graphs.

▶ **Lemma 16.** *Let $c = c(k)$ be a large enough constant, and fix a graph $G = (V, E)$. For any $d \geq cn^{1/k}$, let $V_d = \{v \in V \mid d(v) \geq d\}$. If G is C_{2k}-free, then*

$$|V_d| \leq \begin{cases} O_k(\max(n^{\frac{k-1}{k+1}}, (n^{\frac{k+1}{2k}}/d)^{\frac{2k}{k-1}})), & \text{for odd } k \geq 3, \\ O_k(\max(n^{\frac{k}{k+2}}, (n^{1/2}/d)^{\frac{2k}{k-2}})), & \text{for even } k \geq 4. \end{cases}$$

Here, the notation $O_k(\cdot)$ hides constants that depend on k.

For a graph H and an integer n, let $\text{ex}(H, n)$ be the Túran number of H - the maximum number of edges in an n-vertex H-free graph.

Because the calculations become tedious for general k, we include here a (somewhat informal) proof for 6-cycles ($k = 3$), which conveys the general ideas, and defer the general case to the full version of the paper [12].

Proof of Lemma 16 for $k = 3$. We need to show that $|V_d| \leq O(\max(n^{1/2}, (n^{2/3}/d)^3))$.

Define the following sets of edges:

- $E_d = E \cap (V_d \times V)$: edges adjacent to some node in V_d.
- E_{int}: the internal edges $E_{int} = E \cap (V_d \times V_d)$ of V_d.
- $E_{ext} = E_d \setminus E_{int}$: the external edges $E_{ext} = E \cap (V_d \times (V \setminus V_d))$ adjacent to some node in V_d.

Observe that $|E_d| \geq d \cdot |V_d|/2$, since every node in V_d has degree at least d. On the other hand, since G is C_6-free, so is the subgraph G_d induced on G by V_d; therefore, $|E_{int}| \leq \text{ex}(|V_d|, C_6) = O(|V_d|^{1+1/k})$, where $\text{ex}(N, C_6) = O(N^{1+1/3})$ [2]. Because we required that $d \geq cn^{1/k}$, we have $|E_{int}| = O(|V_d|^{1+1/k}) = O(n^{1/k}|V_d|) \overset{\text{choice of } c}{\leq} (d \cdot |V_d|)/4$. Therefore,

$$|E_{ext}| = |E_d| - |E_{int}| \geq d \cdot |V_d|/2 - d \cdot |V_d|/4 = \Omega(d \cdot |V_d|). \tag{1}$$

In other words, if V_d is large, then the cut between V_d and $V \setminus V_d$ is also large. However, to this cut we can apply Theorem 15: the bipartite graph induced by G on $V_d \times (V \setminus V_d)$ is C_6-free, because all of G is C_6-free; by Theorem 15,

$$|E_{ext}| \leq z(|V_d|, |V \setminus V_d|, C_6) \leq z(|V_d|, n, C_6) = O((|V_d| \cdot n)^{2/3} + n). \tag{2}$$

(We rely on the fact that Zarankiewicz numbers are non-decreasing, because for any a and $b' > b$, an H-free, (a, b)-bipartite graph with e edges is trivially extended to a H-free, (a, b')-bipartite graph with e edges by simply adding nodes on the right side that have no edges.)

Our bound follows from (2), depending on which of the two terms, $(|V_d| \cdot n)^{2/3}$ or n, is larger. If $(|V_d| \cdot n)^{2/3} = O(n)$, then $|V_d| \leq O(n^{1/2})$. Otherwise, from (2) we have $|E_{ext}| = O((|V_d| \cdot n)^{2/3})$, and together with (1), we obtain $|V_d| = O((n^{2/3}/d)^3)$, completing the proof. ◀

The C_{2k}-algorithm from [14] uses the bound on the number of high-degree nodes as follows. First, we search for a C_{2k} that contains a high-degree node: we go over these nodes one after the other, and starting a BFS from each one to check if it participates in a copy of C_{2k}. Subsequently, the high-degree nodes are removed from the graph, together with all their edges. Next, we rely on an observation already used in [11], which is that a C_{2k}-free graph has *arboricity* at most $O(n^{1/k})$; in particular, its vertices can be quickly partitioned into $O(\log n)$ layers $V^1, \ldots, V^\ell$, $\ell = \Theta(\log n)$, such that the number of edges from any node in layer V^i to all higher layers $\bigcup_{j>i} V^j$ is $O(n^{1/k})$. Together with the fact that all high-degree nodes have been removed from the graph, this partition allows us to quickly find any remaining copies of C_{2k} in the graph.

For an integer d, let $V_d = \{v \mid d(v) \geq d\}$ be the set of vertices with degree at least d. Putting both parts together, the running time of the C_{2k} algorithm from [14] is given by:

$$\tilde{O}_k(|V_{n^\delta}| + n^{(k-2)\delta + \frac{1}{k}}), \tag{3}$$

where $\delta \in (0,1)$ is a parameter that determines the threshold for what is considered "high degree". In [14], the value of δ is chosen fairly naïvely: since a C_{2k}-free graph has at most $O(n^{1+1/k})$ edges, for any δ we have $|V_{n^\delta}| \leq O(n^{1+1/k-\delta})$, and balancing the two terms in (3) yields the result.

Our C_{2k} algorithm retains the framework described above, but uses the bound from Lemma 16 to bound $|V_d|$. This allows us to choose a smaller value for δ, lowering the threshold for what we consider a "high-degree node". We obtain the following improved C_{2k}-freeness algorithm. Again, we focus here on the case of 6-cycles, and the full version, which involves even more calculations, appears in the full version of the paper [12].

▶ **Theorem 17.** *C_6-freeness can be solved in $\tilde{O}(n^{3/4})$ rounds in the* CONGEST *model.*

Proof. Recall from (3) that after choosing a degree threshold n^δ, we can eliminate any potential 6-cycle involving a node of V_{n^δ} in $O(|V_{n^\delta}|)$ rounds, and then check the remaining graph in $O(n^{\delta+1/3})$ rounds. Setting $\delta = 5/12$, we have by Lemma 16 that $|V_{n^{5/12}}| = O(n^{3/4})$ and also $n^{\delta+1/3} = n^{3/4}$, yielding the desired running time. ◀

5 Subquadratic Algorithm for Subgraph-Freeness for Any Subgraph

In [14], it was shown that for any $k \geq 1$, there is a subgraph H_k of size $|H_k| = O(k)$ such that randomized H_k-freeness requires $\tilde{\Omega}_k(n^{2-\frac{1}{k}})$ rounds. (The notation $\tilde{\Omega}_k$ hides factors that depend only on k, which we think of as constant.) The bound approaches quadratic as k grows, but in subgraph-freeness, we typically think of k as a constant rather than a growing function. This leads to the question: is there a single subgraph H of some fixed size, such that H-freeness requires $\tilde{\Omega}(n^2)$ rounds?

In this section we give a negative answer, by showing an upper bound that nearly matches the lower bound from [14]:

▶ **Theorem 18.** *For any constant $k \geq 4$ and any subgraph H of size k, there is an algorithm for H-freeness and exact H-counting in $\tilde{O}(n^{2-\frac{2}{3k+1}+o(1)})$ rounds.*

The same upper bound also holds for *induced* subgraph-freeness and counting. Furthermore, the algorithm can be modified to enumerate all copies of H, in $\Theta(n^{2-\Theta(1/k)} + \sqrt{\#H})$ rounds, where $\sqrt{\#H}$ is the number of copies present. We can also show that there exist graphs H which require $\Omega(\sqrt{\#H})$ rounds to enumerate all copies.

Our algorithm begins by decomposing the edges of the graph into two sets, E_m, E_r, with the following properties. Here, $\delta = 1 - \Theta(1/k)$ and $S = n^{\Theta(1/k)}$ are parameters whose exact values will be fixed later.

(1) The graph induced by E_m is composed of maximal connected components $CC = \{R_i\}_{i\in[C]}$ (where C is the number of connected components), which we refer to as *centralized components*. In each centralized component R_i, there is a special *root vertex* $r_i \in V(R_i)$, whose identity is known to all vertices in R_i. Moreover, the root vertex knows all the edges of the subgraph $G\{R_i\}$ (the subgraph induced on G by the vertices $V(R_i)$).

(2) $|E_r| \le n^{2-2\delta}S$, and each vertex in G knows all the edges in E_r.

The decomposition is computed in $\tilde{O}(n^2/S + n^{2-\delta/10} + n^{1+\delta})$ rounds. We give a brief overview, and then explain how to use the centralized components.

5.1 Computing the Centralized Components

The first step in computing the decomposition is to run the decomposition from [9] with different parameters than the ones used in [9]. It returns a partition of the graph into three edge sets, $E = E_{m'} \cup E_{s'} \cup E_{r'}$, where

(a) $E_{s'}$ has bounded arboricity,

(b) $E_{r'}$ is subquadratic in size,

(c) $E_{m'}$ consists of vertex-disjoint high-conductance clusters with high minimum degree.

Next, we divide the clusters of $E_{m'}$ into *equivalence classes*, where two clusters are in the same cluster iff they have many edge disjoint paths between them. We prove that indeed this relation is transitive: if a cluster A has many edge disjoint paths to clusters B and C, then clusters B, C have many disjoint paths between them. The resulting equivalence classes have high connectivity, and therefore, some vertex in each class can learn all the edges in that class in sub-quadratic time. Moreover, the cut separating any two distinct equivalence classes is bounded in size (otherwise they would be the same class).

The algorithm takes E_r be the set of all edges that participate in some cut between two distinct equivalent classes; there are at most $n^{\Theta(1/k)}$ such edges, because the cuts are not too large. Finally, let $E_m = E \setminus E_r$.

5.2 Finding Copies of H

After computing the decomposition, we can immediately detect any copy of the subgraph H whose vertices all belong to the same centralized component R_i, because each such copy is known to the root vertex $r_i \in R_i$. We can also find any copies of H whose edges are contained entirely in E_r, because all nodes know E_r. It remains to find copies of H that include some edges from E_m and some from E_r; that is, copies of H that include at least one vertex from some centralized component, but also some edge from E_r. To find such "split" copies of H, we "guess" which vertices of H should be mapped to a vertex inside some centralized component, and which vertices of H should be mapped to vertices adjacent to edges of E_r, and then verify that we can "stitch together" a complete copy of H by having the root vertices locally check that their centralized component contains the missing pieces and that E_r connects everything properly. This must be done carefully, to ensure that we do not detect false copies of H; we now describe the process in more detail.

Mappings and partial mappings of H. A copy of H can be represented by a mapping $\rho : V(H) \to V(G)$ of the vertices of H to the vertices of G, such that for any $x, y \in V(H)$, if $(x, y) \in E(H)$ then $(\rho(x), \rho(y)) \in G$. We call such a mapping a *good total mapping of H*.

Given a good total mapping ρ of H, we are interested in understanding how the copy of H witnessed by ρ is split between centralized components, so that we can "stitch it together". Let $V_r \subseteq V(G)$ be the set of vertices adjacent to the edges of E_r. We distinguish between two types of vertices:

- *Border vertices:* vertices $x \in V(H)$ such that $\rho(x) \in V_r$. These are vertices serve as potential "stitching points", because they are adjacent to the edges E_r that interconnect the centralized components.
- *Internal vertices:* vertices $x \in V(H)$ such that $\rho(x) \in (\bigcup_i V(R_i)) \setminus V_r$.

(This is a partition, i.e., any vertex of H is either a border vertex or an internal vertex.)

A *border mapping* is a mapping $\sigma : V(H) \to V_r \cup \{*\}$, which specifies for some vertices $x \in V(H)$ a target $\sigma(x) \in V_r$ onto which they are mapped in G, and leaves some vertices unmapped, $\sigma(x) = *$. Intuitively, a border mapping specifies the "interface" between a copy of H whose existence we are trying to verify, and each of the centralized components. We say that a border mapping σ is *good* if there exists a good total mapping ρ of H, such that

- ρ extends σ, that is, for any $x \in V(H)$, if $\sigma(x) \neq *$, then $\rho(x) = \sigma(x)$; and,
- ρ does not add any border vertices, that is, for any $x \in V(H)$ such that $\sigma(x) = *$, we have $\rho(x) \notin V_r$.

Clearly, if G contains a copy of H, then there is a good border mapping, obtained by taking a good total mapping of H and replacing any $\rho(x) \notin V_r$ with $*$.

Our algorithm enumerates over all border mappings, and checks whether each one is good. If we find a good border mapping, we reject, as we have found a copy of H. If we have gone over all border mappings and none are good, we accept. It remains to show that we can efficiently check whether a given border mapping is good, and also that there are not too many border mappings to check.

Checking a border mapping. Fix a border mapping $\sigma : V(H) \to V_r \cup \{*\}$ which is known to all nodes of G, and let us describe how the nodes check whether σ is good, i.e., whether it can be extended into a good total mapping by adding internal vertices.

We call an edge $\{x, y\} \in E(H)$ *external* if $\sigma(x) \neq *, \sigma(y) \neq *$ and $\{\sigma(x), \sigma(y)\} \in E_r$, that is, this edge is mapped outside the centralized components. An edge that is not external is called *internal*, and it includes at least one internal node (possibly two).

Each root vertex locally checks which parts of H it is "responsible for filling in", as follows: we delete from H all the external edges (but not their endpoints), obtaining a collection $H_1, \ldots, H_k$ of connected components – at least two, since we assumed that H is not contained inside any centralized component. Observe that for each $x \in V(H)$ such that $\sigma(x) = *$, there is some i such that $x \in V(H_i)$.

A component H_i is *owned by centralized component R* if there is some internal edge $\{x, y\} \in E(H_i)$ such that $\sigma(x) \in V_r \cap V(R)$, that is, an internal edge that "touches" R. Note that each H_i is owned by exactly one centralized component, and furthermore, if H_i is owned by R, then the internal nodes of H_i *must be filled in* using nodes of R: let us say that a total mapping $\rho : V(H) \to V(G)$ *respects ownership* if for any internal node $x \in H_i$ (for some $i = 1, \ldots, k$), if H_i is owned by centralized component R, then $\rho(x) \in V(R)$.

▶ **Lemma 19.** *For any border mapping σ, any good total mapping ρ that extends σ and does not add any border nodes must respect ownership.*

After deciding which centralized components own which parts $H_1, \ldots, H_k$, the centralized components try to locally complete σ by assigning internal vertices they own to vertices inside the centralized component. More concretely, for each i, the root vertex r whose centralized component R owns H_i looks for a *local mapping for* H_i, $\rho_i : H_i \to V(R)$, such that

- ρ_i agrees with σ on all border vertices in H_i, that is, for any $x \in H_i$ such that $\sigma(x) \in V_r$, we have $\rho_i(x) = \sigma(x)$; and,
- Each edge of H_i is mapped onto some edge of E_m inside R. That is, if $\{x, y\} \in E(H_i)$, then $\{\rho_i(x), \rho_i(y)\} \in E_m$.

We say that a root vertex $r \in V(R)$ is *happy* if, for each H_i owned by R, there is a local mapping ρ_i satisfying the requirements above.

So far, everything we described is done locally, with no communication – all nodes know the edges E_r, and consequently the know V_r; and each root vertex knows all edges of E_m inside its centralized component, so it can check if there is a local mapping ρ_i satisfying the requirements.

Finally, each root vertex announces whether it is happy or not, and the network computes an AND over these answers to check if all root vertices are happy. If all are happy, then σ is a good mapping, and the network rejects. Otherwise, we move on to the next border mapping.

If all root vertices are happy, then we can "piece together" the partial mappings $\sigma, \rho_1, \ldots, \rho_k$ into a total mapping ρ, which we prove is good:

▶ **Lemma 20.** *If there exist local mappings $\rho_1, \ldots, \rho_k$ for $H_1, \ldots, H_k$ (respectively), then G contains a copy of H.*

▶ **Corollary 21.** *Given the decomposition into centralized components, we can check whether G contains a copy of H in $O\left(\left(n^{2-2\delta}S\right)^k + D\right)$ rounds.*

Proof. Since $|E_r| \le n^{2-2\delta}S$, we also have $|V_r| \le n^{2-2\delta}S$, so the number of border mappings is at most $\left(n^{2-2\delta}S\right)^k$. To check each border mapping, we only need to compute an AND over the happiness of each root vertex; using pipelining, we can compute all these ANDs in parallel, in a total of $O\left(\left(n^{2-2\delta}S\right)^k + D\right)$ rounds. ◀

6 Hardness of Proving Lower Bounds For C_{2k}

We give a brief overview of two obstacles on proving strong lower bounds for even-length cycles. All details appear in the full version of the paper [12].

To date, all lower bounds on subgraph-freeness have been shown by reduction from two-party communication complexity, with the players partitioning the network graph between them [11, 14, 21, 8]. The players *simulate* the execution of a distributed algorithm, and the cost of the simulation per round corresponds to the size of the cut between the players' parts. We usually reduce from n-bit set disjointness, which means that we must have (size of the cut) $\cdot$ (number of rounds) $\ge n$ (up to a $\log n$ factor). In [8], it was shown that this approach cannot yield a lower bound greater than $\widetilde{\Omega}(\sqrt{n})$ for 4-cliques, because no matter how we try to partition the graph between the two players, they can solve the K_4-freeness problem using $\widetilde{O}(\sqrt{n} \cdot s)$ bits, where s is the size of the cut between them.

We show an analogous result for 6-cycles, using different ideas: for any fixed partition of the graph between two players, they can solve C_6-freeness using roughly $\sqrt{n} \cdot s$ bits. Our two-party protocol uses the connection to Zarankiewitz numbers that we already exploited in Section 4, to argue that the players can compactly encode paths of length two, three or four that are entirely contained on their side of the graph. This enables the players to "stitch together" a 6-cycle that is split between them, if there is one.

Next, we show that although it seems plausible that the round complexity of C_{2k}-freeness approaches $\Omega(n)$ as $k \to \infty$, proving such a result would imply very powerful circuit lower bounds. In fact, there is an absolute constant $c \in (1/2, 1)$, such that proving any lower bound greater than $\Omega(n^c)$ on *any* particular even-length cycle is already difficult.

We are inspired by a result from [11], where it is shown that the Congested Clique is able to simulate certain types of circuits, which have very large fan-in, efficiently. We are not able to use this result as-is, nor do we simulate the same type of circuit. Instead, we show that **(a)** the problem of C_{2k}-freeness in general networks can be reduced to solving C_{2k}-freeness in networks with high conductance; and **(b)** using the routing scheme of [17, 18], we show that networks with sufficiently high conductance can simulate constant fan-in circuits of depth n^δ, where δ is a constant. Therefore, a lower bound of the form $\Omega(n^c)$, for some absolute constant $c \in (1/2, 1)$, would imply lower bounds on the size of circuits with constant fan-in and depth n^δ. At present, it is still open to find an explicit function that cannot be computed by a circuit of *linear* size and *logarithmic* depth.

References

1. Zvika Brakerski and Boaz Patt-Shamir. Distributed discovery of large near-cliques. *Distributed Computing*, 24(2):79–89, 2011.
2. Boris Bukh and Zilin Jiang. A bound on the number of edges in graphs without an even cycle. *Combinatorics, Probability and Computing*, 26(1):1–15, 2017.
3. Keren Censor-Hillel, Eldar Fischer, Gregory Schwartzman, and Yadu Vasudev. Fast Distributed Algorithms for Testing Graph Properties. In *Distributed Computing: 30th International Symposium, DISC 2016. Proceedings*, pages 43–56, 2016.
4. Keren Censor-Hillel, Petteri Kaski, Janne H. Korhonen, Christoph Lenzen, Ami Paz, and Jukka Suomela. Algebraic Methods in the Congested Clique. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015*, pages 143–152, 2015.
5. Yi-Jun Chang, Seth Pettie, and Hengjie Zhang. Distributed Triangle Detection via Expander Decomposition. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 821–840, 2019.
6. Yi-Jun Chang and Thatchaphol Saranurak. Improved Distributed Expander Decomposition and Nearly Optimal Triangle Enumeration. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019*, pages 66–73, 2019.
7. Norishige Chiba and Takao Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal on computing*, 14(1):210–223, 1985.
8. Artur Czumaj and Christian Konrad. Detecting Cliques in CONGEST Networks. In *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018*, pages 16:1–16:15, 2018.
9. Mohit Daga, Monika Henzinger, Danupon Nanongkai, and Thatchaphol Saranurak. Distributed Edge Connectivity in Sublinear Time. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019*, pages 343–354, 2019.
10. Danny Dolev, Christoph Lenzen, and Shir Peled. “Tri, Tri Again”: Finding Triangles and Small Subgraphs in a Distributed Setting. In *Distributed Computing: 26th International Symposium, DISC 2012. Proceedings*, pages 195–209, 2012.
11. Andrew Drucker, Fabian Kuhn, and Rotem Oshman. On the Power of the Congested Clique Model. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC ’14, pages 367–376, 2014.
12. Talya Eden, Nimrod Fiat, Orr Fischer, Fabian Kuhn, and Rotem Oshman. Sublinear-Time Distributed Algorithms for Detecting Small Cliques and Even Cycles. `https://www.cs.tau.ac.il/~roshman/papers/DISC19_EFFKO.pdf`.

13 Guy Even, Orr Fischer, Pierre Fraigniaud, Tzlil Gonen, Reut Levi, Moti Medina, Pedro Montealegre, Dennis Olivetti, Rotem Oshman, Ivan Rapaport, and Ioan Todinca. Three Notes on Distributed Property Testing. In *31st International Symposium on Distributed Computing (DISC 2017)*, volume 91 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15:1–15:30, 2017.

14 Orr Fischer, Tzlil Gonen, Fabian Kuhn, and Rotem Oshman. Possibilities and Impossibilities for Distributed Subgraph Detection. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures, SPAA 2018, Vienna, Austria, July 16-18, 2018*, pages 153–162, 2018.

15 Pierre Fraigniaud and Dennis Olivetti. Distributed Detection of Cycles. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2017, Washington DC, USA, July 24-26, 2017*, pages 153–162, 2017.

16 Pierre Fraigniaud, Ivan Rapaport, Ville Salo, and Ioan Todinca. Distributed Testing of Excluded Subgraphs. In *Distributed Computing: 30th International Symposium, DISC 2016. Proceedings*, pages 342–356, 2016.

17 Mohsen Ghaffari, Fabian Kuhn, and Hsin-Hao Su. Distributed MST and Routing in Almost Mixing Time. In *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*, pages 131–140, 2017.

18 Mohsen Ghaffari and Jason Li. New Distributed Algorithms in Almost Mixing Time via Transformations from Parallel Algorithms. In *32nd International Symposium on Distributed Computing (DISC)*, pages 31:1–31:16, 2018.

19 Tzlil Gonen and Rotem Oshman. Lower Bounds for Subgraph Detection in the CONGEST Model. To appear in OPODIS 2017, 2017.

20 Taisuke Izumi and François Le Gall. Triangle Finding and Listing in CONGEST Networks. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC '17, pages 381–389, 2017.

21 Janne H. Korhonen and Joel Rybicki. Deterministic Subgraph Detection in Broadcast CONGEST. In *21st International Conference on Principles of Distributed Systems, OPODIS 2017, Lisbon, Portugal, December 18-20, 2017*, pages 4:1–4:16, 2017.

22 Assaf Naor and Jacques Verstraëte. A note on bipartite graphs without 2k-cycles. *Combinatorics, Probability and Computing*, 14(5-6):845–849, 2005.

23 Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. On the Distributed Complexity of Large-Scale Graph Computations. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures, SPAA 2018, Vienna, Austria, July 16-18, 2018*, pages 405–414, 2018.

24 Jeanette P. Schmidt, Alan Siegel, and Aravind Srinivasan. Chernoff-Hoeffding Bounds for Applications with Limited Independence. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '93, pages 331–340, 1993.

25 Michael Szell and Stefan Thurner. Measuring social dynamics in a massive multiplayer online game. *Social Networks*, 32(4):313–329, 2010.

26 Jacques Verstraëte. Extremal problems for cycles in graphs. In *Recent Trends in Combinatorics*, pages 83–116. Springer, 2016.

27 Stanley Wasserman and Katherine Faust. *Social Network Analysis: Methods and Applications*. Structural Analysis in the Social Sciences. Cambridge University Press, 1994.

28 Duncan J. Watts. Collective dynamics of 'small-world' networks. *Nature*, 393:440, 1998.

A Distributed Algorithm for Directed Minimum-Weight Spanning Tree

Orr Fischer
Computer Science Department, Tel-Aviv University, Israel
orrfischer@mail.tau.ac.il

Rotem Oshman
Computer Science Department, Tel-Aviv University, Israel
roshman@mail.tau.ac.il

Abstract

In the *directed minimum spanning tree* problem (DMST, also called *minimum weight arborescence*), the network is given a root node r, and needs to construct a minimum-weight directed spanning tree, rooted at r and oriented outwards. In this paper we present the first sub-quadratic DMST algorithms in the distributed CONGEST network model, where the messages exchanged between the network nodes are bounded in size. We consider three versions: a model where the communication links are bidirectional but can have different weights in the two directions; a model where communication is unidirectional; and the Congested Clique model, where all nodes can communicate directly with each other.

Our algorithm is based on a variant of Lovász' DMST algorithm for the PRAM model, and uses a distributed single-source shortest-path (SSSP) algorithm for directed graphs as a black box. In the bidirectional CONGEST model, our algorithm has roughly the same running time as the SSSP algorithm; using the state-of-the-art SSSP algorithm, we obtain a running time of $\widetilde{O}(\min(\sqrt{nD}, \sqrt{n}D^{1/4} + n^{3/5} + D))$ rounds for the bidirectional communication case.

For the unidirectional communication model we give an $\widetilde{O}(n)$ algorithm, and show that it is nearly optimal. And finally, for the Congested Clique, our algorithm again matches the best known SSSP algorithm: it runs in $\widetilde{O}(n^{1/3})$ rounds.

On the negative side, we adapt an observation of Chechik in the sequential setting to show that in all three models, the DMST problem is at least as hard as the (s,t)-shortest path problem. Thus, in terms of round complexity, distributed DMST lies between single-source shortest path and (s,t)-shortest path.

2012 ACM Subject Classification Networks → Network algorithms; Theory of computation → Distributed algorithms

Keywords and phrases Distributed Computing, Directed Minimum Spanning Tree, Minimum Arborescence, CONGEST

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.16

Related Version A full version of the paper is available at `https://www.cs.tau.ac.il/~roshman/papers/DISC19_FO.pdf`.

Funding *Orr Fischer*: Research supported by the Israeli Centers of Research Excellence (I-CORE) program (Center No. 4/11) and the Laura Schwarz-Kipp Institute of Computer Networks.
Rotem Oshman: Research supported by the Israeli Centers of Research Excellence (I-CORE) program (Center No. 4/11).

1 Introduction

Finding a lightweight spanning subgraph of a network is among the most fundamental problems in distributed computing. The classical example is the minimum-weight spanning tree (MST) problem, which has received extensive attention: its round complexity in the CONGEST model was tightly characterized in a series of papers (e.g [14, 26, 15, 8, 9, 22, 17,

© Orr Fischer and Rotem Oshman;
licensed under Creative Commons License CC-BY

23, 10]). Generalizations, such as minimum-weight k-vertex-connected and k-edge connected subgraph, have also been studied (e.g [10, 5, 31]). To date, almost all distributed algorithms for MST and related problems have been for *undirected* graphs, with *symmetric* edge weights. However, in many settings, the cost associated with an edge is not necessarily symmetric: for example, in a wireless network, the energy required to send a message to a specific node can depend on contention and noise in that node's vicinity, and in peer-to-peer cellular phone mesh networks, the price of communicating across a given link could be dictated by market forces. If we have a single node that needs to repeatedly broadcast to the entire network, or to collect information from the entire network, can we quickly find a cheap spanning tree – oriented downwards (or upwards) – allowing it to do so?

The *directed minimum-weight spanning tree* (DMST) problem asks exactly this question: we have a weighted graph G, where edge weights are not necessarily symmetric, and a fixed root node r. Our goal is to construct a minimum-weight directed spanning tree, rooted at r and oriented downwards (or upwards).

Although the DMST problem has been extensively studied in the sequential setting [32, 7, 2, 6, 28, 29, 13], to date there has not been a distributed solution for DMST that runs quickly and does not use a lot of communication. In fact, prior to our work, no non-trivial (i.e., sub-quadratic) algorithm for the CONGEST model was known. In this paper we give distributed DMST algorithms for three variants of the CONGEST model: (a) undirected communication networks with asymmetric edge weights; (b) directed communication networks; and (c) the Congested Clique model, where the communication network is the complete graph (a clique).

Undirected MST is known to require $\tilde{\Theta}(\sqrt{n} + D)$ rounds [30, 10]. Clearly, we cannot hope for DMST to require less, as MST is a special case of DMST. Furthermore, in some scenarios (e.g., sequential dynamic graph algorithms), DMST is believed to be significantly harder than MST. Surprisingly, we show that when the underlying communication network is undirected and has a small diameter, DMST is no harder than MST. In fact, we show that in undirected networks, DMST essentially "reduces" to directed single-source shortest path (SSSP), so that up to a logarithmic factor, its round complexity is bounded from above by the running time of the best SSSP algorithm that can handle asymmetric weights (currently [12]). On the other hand, we show that DMST is no easier than (s, t)-shortest path – this is already known in the sequential setting (see Section 6), and we show that it also holds in all three variants of the CONGEST model. Therefore, DMST's round complexity is sandwiched between SSSP and (s, t)-shortest path.

Background. The best sequential algorithm for DMST is Gabow et al.'s implementation of Edmonds' algorithm [7, 13]. It performs a series of *contractions*, where every vertex $v \neq r$ deducts the weight of its minimum-weight incoming edge from all its incoming edges, and then each zero-weight directed cycle is contracted into a single vertex. Eventually, we are left with a zero weight tree; the weight of the DMST is then given by the sum of all the weights deducted during the algorithm's run (See Section 4 for details). Actually *finding* the DMST is not immediate, and requires recursively undoing the contractions and carefully adding edges to the DMST at each step. (Counter-intuitively, the lightest incoming edge of any given node does not necessarily belong to a DMST, and in fact, even the lightest edge in the entire graph might not belong to it.)

The drawback of Edmonds' algorithm in a parallel setting is that it may require $n - 1$ contractions to contract the entire graph, and the contractions are not easy to parallelize. In [24], Lovász gave a PRAM algorithm that "speeds up" this process, and contracts the entire graph in $O(\log n)$ parallel steps. In CONGEST, Lovász' algorithm cannot be implemented

efficiently as-is, for several reasons – including the fact that it uses all-pairs shortest path (APSP) as a subroutine (APSP requires linear time in CONGEST [1]), and that certain steps of the algorithm would lead to too much congestion if we try to implement them in CONGEST. We modify Lovász' algorithm to obtain a variant that lends itself to an efficient distributed implementation, and then give implementations for the three variants of CONGEST. The implementations overcome several challenges that are not encountered in undirected MST, such as the fact that in each step we need to run SSSP inside many disjoint subgraphs, but each component can have a diameter that is much larger than the diameter of the network as a whole. If we called SSSP directly inside each component, our running time would depend on the largest diameter encountered during the run, which could be linear in the worst case. We show how to overcome this difficulty in Section 5.

Our results. We give one "meta-algorithm" for DMST, and then implement it in the three models we consider (undirected, directed, and Congested Clique). For the bidirectional CONGEST model and the Congested Clique, we show that given an efficient algorithm for single-source shortest paths (SSSP), we can find a DMST in roughly the same running time. Specifically, let $T(n, D)$ be the time required in CONGEST to compute SSSP in undirected graphs of size n and diamter D, with non-negative, asymmetric integer weights, and let $\mathcal{A}_{\text{SSSP}}$ be an SSSP algorithm with running time $T(n, D)$. We prove:

▶ **Theorem 1** (Informal). *There is a DMST algorithm for undirected* CONGEST *with asymmetric weights that runs in $\widetilde{O}(T(n, D))$ rounds. Moreover, the DMST algorithm is deterministic if $\mathcal{A}_{\text{SSSP}}$ is deterministic.*

We take extra care to ensure that our "reduction" from DMST to SSSP be *deterministic*, so that if in the future an efficient deterministic SSSP algorithm is discovered, we can use it to get a deterministic DMST algorithm.

Plugging in the randomized Las-Vegas SSSP algorithm of [12], we obtain the following algorithm for the undirected CONGEST model with asymmetric edge weights:

▶ **Theorem 2.** *In the undirected* CONGEST *model with asymmetric weights, there is a randomized DMST algorithm that always succeeds, and requires $\widetilde{O}(\min(\sqrt{nD}, \sqrt{n}D^{1/4} + n^{3/5} + D))$ rounds in expectation.*

For small diameter networks, $D = O(\text{polylog}(n))$, our algorithm is optimal up to polylogarithmic factors, and nearly matches the lower bound for *undirected* MST [30]. For larger diameter, we can also write the running time as $\widetilde{O}(n^{2/3} + D)$, a slightly weaker bound than the one stated in Theorem 2. Since our algorithm calls the SSSP algorithm as a black box, any improvement in SSSP will yield an improved DMST algorithm as well.

A similar result holds for the Congested Clique. At present, the best SSSP algorithm for that model runs in $\tilde{O}(n^{1/3})$ rounds [3], and so we obtain an $\tilde{O}(n^{1/3})$-round DMST algorithm for the Congested Clique. For the directed communication model, we give a deterministic algorithm with running time $\widetilde{O}(n)$, and we show that this is tight (up to a logarithmic factor). The algorithm and the lower bound assume that the weight of each edge (u, v) is known only to its destination v, and that G is strongly connected. These results are described in full version of the paper [11].

As Theorem 2 shows, in the undirected CONGEST model, the DMST problem is no harder than single-source shortest path. Is the converse true? For the sequential setting, this is conjectured to hold, and Chechick showed [4] that DMST is at least as hard as the (s, t)-shortest path problem. We give a reduction that allows the proof from [4] to work in the distributed setting, showing that DMST is no easier than (s, t)-shortest path in all three distributed models we consider.

For lack of space, we only give a high-level overview of the algorithm for the undirected case, and defer many technical details – as well as pseudo-code and proofs of correctness – to the full version of the paper [11]. The other two models (directed networks and the Congested Clique) are also relegated to the the full version of the paper. Finally, we focus here on computing the *weight* of the DMST, and defer the details of how to find the DMST edges (which is requires some details) to the last section of the paper.

We note that our algorithm naturally extends to approximating DMST using an approximate SSSP algorithm for directed graphs with non-zero weights. Using this extension, a c-approximation directed SSSP algorithm yields a $c^{\log n}$-approximation of the DMST (meaning we require an $(1+\frac{1}{\Omega(\log n)})$-approximate SSSP algorithm in order to get a constant or sub-constant DMST approximation using this method). The best known $(1+\epsilon)$-SSSP approximation algorithm for directed graphs in CONGEST has round complexity of $\tilde{O}((\sqrt{n}D^{1/4}+D)/\epsilon)$ [12], which yields an $(1+\frac{1}{\operatorname{polylog} n})$-approximation of the DMST in $\tilde{O}(\sqrt{n}D^{1/4}+D)$ rounds. We defer the details to the full version of the paper.

2 Related Work

Distributed MST is one of the most fundamental problems in CONGEST, with a wide range of works, a very short subset of which include [14, 26, 15, 8, 9, 22, 17, 23, 10]. In particular, Ghaffari et al.[15] gave a simple MST algorithm using a framework called *low-congestion shortcuts*. This framework also serves as the basis for our DMST algorithm, as it allows to handle connected components that grow too large for their nodes to communicate with each other directly. Our algorithm also uses procedures from [19, 10] to deterministically decompose a directed tree into few components with relatively small diameter. Several lower bounds were shown by [27, 8, 30], proving that in the CONGEST model, finding the MST's weight takes $\widetilde{\Omega}(\sqrt{n}+D)$ rounds ,even for any approximation factor of up to $poly(n)$.

Minimum Directed MST (or Minimum weight Arborescence) had been extensively studied in the sequential model. The first algorithms for DMST in the sequential setting were independently found by [32, 7, 2]. A faster implementation was given by Tarjan [6], which included ideas from [28, 29]. The most efficient known implementation of Edmonds' algorithm in the sequential setting is due to [13], with running time $O(m+n\log n)$.

A parallel NC algorithm for DMST was given by Lovász [24]. Humblet [21] showed a distributed $O(n^2)$ round algorithm for DMST with message complexity $O(n^2)$. To our knowledge, ours is the first DMST algorithm for CONGEST that has better than the trivial round complexity of $O(n^2)$.

Our algorithm uses a directed single source shortest path algorithm as a black-box. Recently, two such algorithms were developed [16, 12]. The best known running time for for both directed and undirected graphs is $\widetilde{O}(\min(\sqrt{nD}, \sqrt{n}D^{1/4}+n^{3/5}+D))$ due to Nanongkai et al. [12]. In [12] an $(1+o(1))$-approximation in time $\widetilde{O}(\sqrt{n}D^{1/4}+D)$ for the directed case was shown. In the undirected case, [20] gave a deterministic $(1+o(1))$-approximation in time $\widetilde{O}(n^{1/2+o(1)}+D^{1+o(1)})$. In the Congested Clique, Censor-Hillel et al.[3] gave a $\widetilde{O}(n^{1/3})$ APSP algorithm for directed graphs based on algebraic methods.

3 Preliminaries

Let $G=(V,E,w_G)$ be a weighted directed graph, with a special *root* vertex $r \in V$. We assume throughout that r has a directed path to every node in G. We construct a spanning tree rooted at r and oriented downwards. (To obtain a tree oriented upwards towards

r, we can simply reverse all edge directions.) For convenience, if $(u, v) \notin E$, then we set $w_G(u, v) = \infty$. Also, given a vertex set $A \subseteq V$, we denote by $G(A)$ the subgraph induced on G by A. We sometimes abuse notation by writing $u \in H$ when u is a vertex of a subgraph H.

We assume w.l.o.g. that edge weights are integers in the range $[0, ..., \text{poly}(n)]$. This is not essential; negative weights and larger weights are easily handled, although if weights require more than $O(\log n)$ bits to represent, the SSSP algorithm will use more rounds.

For two nodes u, v, let $\text{dist}_G(u, v)$ be the weight of the shortest path from u to v according to the weight function w_G. Given a subgraph H of G and two nodes $u, v \in H$, we let $\text{dist}_H(u, v)$ denote the weight of the shortest path *using only vertices of* H from u to v. For a subgraph H, let $\text{In}(H) = \{(u, v) \in E | v \in H \wedge u \notin H\}$ be the set of edges entering H.

4 Overview of the Algorithm

In this section we give a high-level overview of our DMST algorithm, which is based on Edmonds' and Lovász' algorithms.

As it runs, the algorithm performs *contractions*, where a set of vertices is merged into one *super-vertex*. Here we describe the "meta-algorithm" that runs on the graph of super-vertices, and later we will show how this meta-algorithm is implemented on the actual network (where, of course, we cannot merge nodes).

The active edges. Throughout its run, the algorithm maintains a set of zero-weight directed edges, denoted H, with the property that every (super-)vertex except r has in-degree 1 in H.

To initialize H, each node v chooses a minimum-weight incoming edge (u, v), deducts its weight from all incoming edges, and adds (u, v) to H. (If there is more than one incoming edge with the minimum weight, then we choose arbitrarily.)

The weakly-connected component of H that contains the root is called the *root component*. The remaining weakly-connected components of H are called *active components*, and denoted $H_1, \ldots, H_k$. Since the in-degree in H is 1, each active component is a directed cycle, with trees rooted at some of the cycle's vertices and oriented outwards (see Fig. 2). We abuse notation by thinking of each H_i as both a set of edges and as a graph (the weakly-connected component). We let $C(H_i)$ denote the directed cycle that "lies at the heart" of the active component H_i.

The following property is helpful when trying to determine which vertices belong to a given active component: if we know some vertex v that lies on the cycle $C(H_i)$, then the vertices of H_i are exactly those vertices reachable from v along the directed edges of H.

Edmonds' contractions. Edmonds' algorithm makes a series of steps, where in each step,

(1) Each vertex v deducts the weight of its minimum-weight incoming edge from all its incoming edges, and remembers the weight it subtracted. (We must connect v to the DMST by *some* incoming edge, so we will pay at least the weight of its lightest incoming edge.)

(2) Each vertex adds one zero-weight incoming edge to H.

(3) Any newly-created zero-weight directed cycles in H are contracted into a single vertex. (This does not change the weight of the DMST.)

Eventually, we are left with only the root component, on which the H edges induce a directed spanning tree of weight zero. The weight of the DMST is then given by the total weight subtracted by all the nodes during the run. Then, we must "undo" the constructions and compute the edges of the DMST; we defer this part to the end of the paper, and focus for now on computing the weight of the DMST.

Each step of Edmonds' algorithm contracts at least two vertices, but unfortunately, it does not necessarily merge each active component with another active component: an active component might spend many steps contracting nested cycles of inner vertices, one after the other. While each step reduces the number of vertices by at least 1, we might require as many as n steps to contract the entire graph.

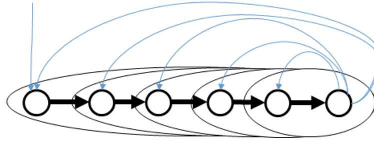


Figure 1 An active component in which Edmonds' algorithm contracts only two vertices at a time. In the worst case $\Omega(n)$ contractions occur before the active component is merged with another component.

Lovász' algorithm can be viewed, somewhat inaccurately, as a way to "jump ahead": roughly speaking, instead of spending a lot of time contracting nested cycles inside an active component, Lovász finds the first edge *coming in from outside the component* that would be added to H, and then performs in one fell swoop all the nested contractions leading up to that point. (This is not accurate, as we explain below; one step of Lovász cannot always be decomposed into steps of Edmonds.) After $O(\log n)$ "mega-steps", we are left with only the root component, and then we are done. See the full version of the paper [11] for a more detailed description of Lovász' algorithm.

The "mega-steps" of Lovász are difficult to implement in CONGEST: in each step, the algorithm computes all-pairs shortest-paths (APSP, which can be solved efficiently in PRAM), and it finds paths that may cut across many active components, leading to congestion. We give a less eager mechanism for speeding up Edmonds' algorithm, which is quite similar to Lovász but can be performed in parallel on all the active components in CONGEST; essentially, we show that the steps of Lovász' algorithm can be *confined inside the active components* without cutting across them, while preserving correctness and the fast running time.

Our modified meta-algorithm. Our meta-algorithm is obtained from Edmonds by asking: "what contractions would Edmonds' algorithm make inside an active component H_i before it adds to H an *incoming* edge of H_i, thereby merging it with another component?" We would like to jump ahead to that point.

Recall that Edmonds selects at each step the minimum-weight incoming edge of a node, adds it to H, and (eventually) contracts the resulting zero-weight cycle. It turns out that as it slowly consumes nodes inside H_i and eventually some node outside H_i, Edmonds implicitly finds the lightest path from a node outside H_i that *immediately enters* H_i and stays inside H_i until it arrives at some node of the cycle $C(H_i)$ (see Fig. 2); i.e., a path of the form $u, v_1, \ldots, v_k$ such that (a) $u \notin H_i$, (b) we have $v_j \in H_i \backslash C(H_i)$ for each $j = 1, \ldots, k$, and finally, (c) $v_k \in C(H_i)$.

Let β be the weight of this path. Edmonds does not progress *only* along the path; it contracts nodes inside H_i that can reach the cycle $C(H_i)$ with paths of increasing weight, until the weight reaches β. Our meta-algorithm finds the edge (u, v_1) and the weight β, and contracts all nodes $x \in H_i$ that have $\text{dist}_{H_i}(x, C(H_i)) \leq \beta$ (including v_1). (Lovász also computes β, but it does it differently: it uses all-pairs shortest-paths to find the shortest distance from any node outside H_i to the cycle $C(H_i)$, without insisting that the path have the form we described above. As a result, Lovász may find paths that start at a node u,

wander outside H_i for a while (along zero-weight edges), enter H_i and leave it, and eventually enter H_i "for good" and go to the cycle $C(H_i)$. Lovász then makes a more aggressive contraction, merging all the nodes visited by such a path of weight at most β.)

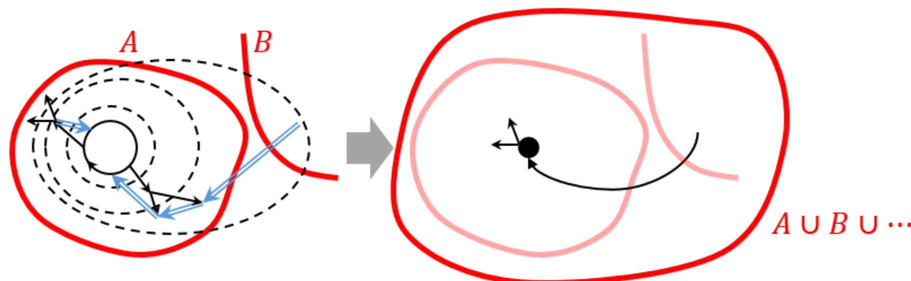


Figure 2 Our algorithm performs in one step three steps of Edmonds' algorithm: it contracts the zero-weight directed cycle of component A and makes two more contractions, finally adding an incoming edge of A to H. Component A then merges with B, and possibly with other components. Edges initially in H are shown as solid lines, edges that are added to H during Edmonds are shown as double lines.

We now give a more formal description. In each step of our meta-algorithm, we find in parallel for each active component H_i an incoming edge $(u, v) \in \text{In}(H_i)$, with $v \in H_i$ and $u \notin H_i$, that minimizes the internal distance to the cycle, $\beta(u, v) := w(u, v) + \text{dist}_{H_i}(v, C(H_i))$. (Recall that $\text{dist}_{H_i}(v, C(H_i))$ is the distance *inside* H_i from node v to any node of the cycle $C(H_i)$; only paths using edges of H_i can be used.)

For an active component H_i, let $\beta_i = \min_{(u,v)\in\text{In}(H_i)} \beta(u, v)$ be the "minimum entering distance" associated with H_i. This is the weight that would be subtracted by Edmonds' algorithm in all the contractions internal to H_i, plus the first step that connects H_i to another active component.

Our algorithm finds an edge (u, v) that has $\beta(u, v) = \beta_i$ (that is, an edge that minimizes $\beta(u, v)$). Then, we contract the zero-weight cycle $C(H_i)$, *together with all nodes inside* H_i that have distance at most $\beta(u, v)$ to the cycle $C(H_i)$. Formally, the set of nodes we contract into one super-vertex is given by

$$U_i := \{v \in H_i \mid \text{dist}_{H_i}(v, C(H_i)) \le \beta_i\} . \tag{1}$$

We represent a super-vertex as the set of all original graph vertices that were merged into it; merging super-vertices means replacing them by their union.

For the new super-vertex S, we update the weights in the contracted graph (in which S is a vertex):

$$w'(x, y) = \begin{cases} \min_{z\in S} \beta(x, z) - \beta_i & \text{if } y = S, \\ \min_{z\in S} w(z, y) & \text{if } x = S, \\ w(x, y) & \text{otherwise.} \end{cases}$$

After the contraction, the incoming edge (u, S), which replaces (u, v) (the edge that had $\beta(u, v) = \beta_i$) and now has weight zero, is added to H. This causes H_i to merge with the active component to which u belongs (see Fig. 2).

Because every active component merges with another active component in each iteration, the number of components is reduced by at least half, and therefore after $O(\log n)$ iterations, the edge set H has only one weakly-connected component – the root component. At this point, the weight of the DMST is computed by summing all the β_i's that were subtracted during the entire run, and the algorithm terminates.

In the full version of the paper [11], we prove that unlike Lovász' algorithm, each step of our meta-algorithm can be decomposed into a series of Edmonds' contractions, and we give an example showing the difference between Lovász's algorithm and our variant.

5 Implementation in CONGEST

In this section we explain, on a high level, how we translate our meta-algorithm to the CONGEST model. We start by introducing the main ingredients that go into the implementation.

The meta-graph and the physical graph. We refer to the "real" nodes of the communication network as *physical vertices* or *physical nodes*. Super-vertices are simply sets of physical vertices, but each super-vertex has a unique identifier, which is the ID of some physical vertex in it. We often conflate a super-vertex with its ID. Let $\mathcal{S}$ be the set of all super-vertex IDs (as we said, these are simply IDs from V, but for clarity we use different notation).

During the run of our algorithm, each physical vertex v keeps track of $sId(v)$, the ID of the super-vertex that contains it in the meta-graph. Node v also knows which of its physical edges correspond to meta-edges in H: that is, for each physical edge $\{v, u\}$, node v knows whether or not $(sId(v), sId(u)) \in H$.

Given a physical network graph $G = (V, E, w)$ and a mapping $sId : V \to \mathcal{S}$ of physical nodes onto super-vertices, the meta-graph that corresponds to G and sId is a *multi-graph*, where two super-vertices $S_1, S_2 \in S$ are connected by all the edges that connect physical vertices $(u, v) \in E$ such that $u \in S_1, v \in S_2$. (Although our modified algorithm above is stated for graphs rather than multi-graphs, it is easy to see that its correctness translates immediately to multi-graphs as well.) For each super-vertex $S \in \mathcal{S}$, there is a single incoming meta-edge (T, S) in H (recall that all nodes have in-degree exactly 1 in H). The meta-edge (T, S) may correspond to many physical edges; the algorithm chooses one such edge, $(u, v) \in E$ such that $u \in T$ and $v \in S$, and defines $\text{entry}(S) = v$ to be "the physical entry-point of S".

Soft contractions. Since we cannot contract vertices of the communication network, we replace contractions with *soft contractions*, which have the same effect but change only the weight function and the super-vertex mapping.

▶ **Definition 3** (Soft contraction). *Fix a physical graph $G = (V, E, w_G)$, a mapping $sId : V \to \mathcal{S}$ of physical vertices to their super-vertices, a set $H \subseteq \mathcal{S}^2$ of zero-weight directed meta-edges, an active component H_i, and a set $A \subseteq \mathcal{S}$ of super-vertices to contract. Define $G_{\sim A} = (V, E, w_{G_{\sim A}})$ to be the physical graph with the same vertices and edges as G, but with the following weight function:*

$$w_{G_{\sim A}}(u, v) = \begin{cases} w_G(u, v) + \text{dist}_{G(A)}(v, C(H_i)) - \beta_i & \text{if } u \in V \setminus A \text{ and } v \in A, \\ 0 & \text{if } u, v \in A \text{ and } (u, v) \in E, \\ w_G(u, v) & \text{otherwise.} \end{cases}$$

The soft contraction *operation updates the weights as above, and replaces the mapping sId with sId', where $sId'(v) = sId(v)$ if $v \notin A$, and $sId'(v) = A.id$. The id of A is the id of some "leader" vertex $v \in A$.*

Intuitively, a soft contraction is the same as the meta-step we defined in Section 4, but instead of merging vertices, it simply zeroes out the weight of the edges between them. We prove that if we take $A = U_i$ (as defined in (1) above), then the soft contraction operation is equivalent to the meta-step we defined in Section 4, and decreases the weight of the DMST by β_i.

Small and large components. As usual in MST algorithms, after we perform some meta-steps of the algorithm, some super-vertices may become so large that we cannot afford for their physical nodes to communicate with each other directly. We resolve this in the usual way (see, e.g., [15, 10]): super-vertices are classified into "small" super-vertices, which comprise at most $\sqrt{n}$ physical nodes, and "large" super-vertices, comprising more than $\sqrt{n}$ nodes. The small super-vertices are small enough that we can compute on them directly (in parallel). As for large super-vertices, there are at most $\sqrt{n}$ of them, and the entire network helps them carry out their computation. For example, if we have large super-vertices $S_1, \ldots, S_k$, and we want each physical vertex of S_i to learn some value x_i, then we will propagate all values $x_1, \ldots, x_k$ throughout the entire network, and each physical vertex $v \in S_i$ will pick out the value x_i it needs to learn.

We remark that unlike undirected MST, in our case there is a distinction between a *super-vertex* and an *active component*. (In distributed implementations of Boruvka's undirected MST algorithm, there are super-vertices, but there is no notion of "active component".) In addition to computing on all the super-vertices in parallel, our algorithm also carries out steps on the *active components* in parallel, but an active component consists of many super-vertices, some small and some large. This presents some complications compared to the undirected case.

Centers. A key part of our algorithm is concerned with finding some super-vertex that lies on the cycle $C(H_i)$ of an active component. This cycle may consist of any number of super-vertices, themselves comprising many physical-vertices. To find a super-vertex on the cycle, we "chop up" the cycle into more manageable parts: we select a *center set*, a set of $\tilde{O}(\sqrt{n})$ super-vertices (always including the root super-vertex), with the property that for any H-path $S_1, \ldots, S_k$ of super-vertices, if the total number of physical vertices in $S_1, \ldots, S_k$ is at least $\sqrt{n}$, then at least one super-vertex S_i is a center.

Centers are often used in shortest-path computations (e.g., [12, 16, 25] in the CONGEST model, and many other examples in dynamic algorithms and distance oracles), but here we use them in a non-standard way: we construct a *center graph* representing the reachability relation between centers, and use this graph to find the cycle $C(H_i)$ and determine which super-vertices are reachable from it.

Running SSSP on many disjoint components in parallel. During our algorithm we encounter the following scenario: we have a collection of vertex-disjoint connected subgraphs $G_1, \ldots, G_k \subseteq G$, with one marked node $v_i \in G_i$ in each component, and also some external node $r \notin \bigcup_i G_i$. The diameter of the entire graph G is D, but the diameter of each G_i can be arbitrarily large. We wish to compute, *in parallel* for all i, the distances $\text{dist}_{V_i}(v_i, u)$ (that is, the distance from v_i to each node inside its component V_i, using only nodes of V_i).[1]

[1] The keen-eyed reader might notice that the directions here are reversed – in Section 4 we wanted distances *to* a node of $C(H_i)$, and now we ask for distances *from* some node to all others in the component. We handle this by reversing all edge directions.

Moreover, we want to "pay" only in terms of the diameter D of the entire network, not the diameters of the individual components. This rules out running a separate SSSP instance inside each component using only its internal edges.

Our solution is to simulate *one* execution of SSSP on a "virtual network" G', defined as follows. The vertices of G' are the vertices of G, but we also add, for each $v \in V(G)$, a "shadow vertex" v'. The edges of G' are

(a) all edges that are internal to some subgraph G_i, with their original weights; (b) the "shadow copies" (u', v') of all edges in E, with weight zero; (c) for each marked node v_i, we add a zero-weight edge (v_i', v_i) from v_i's shadow to v_i, and also an edge (v_i, v_i') in the opposite direction, with "infinite" (or sufficiently large) weight.

The network G' can be simulated efficiently by the nodes of G, by having each node simulate itself and its shadow. Note that the edges we added allow for such a simulation; for example, two shadow nodes only need to communicate in G' if their corresponding "real nodes" can communicate in G. Also, $\mathrm{diam}(G') \le 2\,\mathrm{diam}(G) + 1$.

Now, to simultaneously compute all the distances $\mathrm{dist}_{V_i}(v_i, u)$, we simulate a call to SSSP from node r', the shadow of r, in G'. A lightest path from r' to a node $u \in V_i$ traverses the shadow network from r' to v_i' at zero cost, then moves to v_i with no cost, and then traverses from v_i to u inside the "real" copy of G_i. Thus, the distance from r' to $u \in V_i$ in G' is exactly $\mathrm{dist}_{V_i}(v_i, u)$. See full paper for details.

5.1 The Algorithm

We now give a more detailed description of our algorithm (while still omitting many technical details). The algorithm runs in $O(\log n)$ iterations. At the beginning of each iteration, each node $v \in V$ knows an identifier $sId(v)$ for its super-vertex (initially, $s(v) = v$), it knows which of its edges correspond to meta-edges in H, and it knows whether or not it is part of the root component.

Nodes do not necessarily know which active component they belong to at any given moment; the first part of each iteration of our algorithm is concerned with finding the current active components, after some of them were merged at the end of the previous iteration. Nevertheless, it is convenient to think of the algorithm as "operating in parallel" on all the active components.

Each iteration proceeds as follows, in parallel for each active component H_i:

(1) We find some super-vertex $c(H_i) \in C(H_i)$ that lies on the cycle of H_i, and disseminate the ID of $c(H_i)$ to all physical nodes in H_i. In particular, we must determine which super-vertices belong to H_i. This is described in Section 5.2.

(2) We compute shortest paths from all super-vertices of H_i to $C(H_i)$: this is done by a single call to SSSP, as described above, using $c(H_i)$ as the marked node in component H_i. We use reverse edge weights, so that instead of computing shortest paths *from* $c(H_i)$ we compute shortest paths *to* $c(H_i)$. Note that since $C(H_i)$ is a cycle of zero-weight edges, the distance to $c(H_i)$ is also the distance to all nodes of $C(H_i)$.

(3) We find an incoming edge $e_i = (u, v) \in \mathrm{In}(H_i)$ that minimizes the "entering distance", $\beta(u, v) = w(u, v) + \mathrm{dist}_{H_i}(v, C(H_i))$, and disseminate e_i and $\beta_i = \beta(e_i)$ to all nodes of H_i. This is done using the small component/large component methodology, but some care is needed (as done in [15, 18]. see procedure `LearnMin` in the full version for details [11]).

(4) Finally, having computed β_i and $e_i = (u, v) \in \mathrm{In}(H_i)$, we soft-contract H_i with threshold β_i, "virtually merging" all super-vertices with distance at most β_i to $C(H_i)$ into one super-vertex. The ID of the new super-vertex is set to $c(H_i)$. We add edge e_i to H, which has the implicit effect of merging H_i with another active component.

After $O(\log n)$ iterations, no active components remain, and we have only the root component. We now compute a spanning tree of the network graph, and use it to sum the values of β_i subtracted throughout the algorithm. The root of the DMST returns this value as the weight of the DMST.

5.2 Finding A Cycle Super-Vertex and Identifying the Active Component

In this section we show how to find, for each active component H_i, some super-vertex $c(H_i)$ on the cycle $C(H_i)$. When we begin this part of the algorithm, the physical nodes know which of their edges are in H, but they do not know which active component (i.e., which weakly-connected components of H) they belong to. Part of our goal is to identify the boundaries of the active components, in preparation for finding a minimum-distance incoming edge of each active component; this is accomplished by disseminating $c(H_i)$ to all nodes that can be reached from the cycle $C(H_i)$ along paths of H-edges. Thus, $c(H_i)$ serves as an *active component ID*, which all physical nodes of H_i agree on.

As we said above, in order to identify long cycles and paths, we cut them into shorter pieces by choosing a set of *centers*. Formally, we need the following property:

▶ **Definition 4.** *A set of super-vertices $\mathcal{T} \subseteq \mathcal{S}$, which includes the root super-vertex, is said to be a* good center set *if $|\mathcal{T}| \leq 4\sqrt{n}$, and for any H-path $S_1, \ldots, S_k$, if $|\bigcup_{i=1}^{k} S_i| \geq \sqrt{n}$ (that is, if $S_1, \ldots, S_k$ together contain at least $\sqrt{n}$ physical vertices), then $\mathcal{T}$ includes some super-vertex S_i.*

A good center set can be constructed deterministically in a very similar manner to either the star-decomposition of [19], or using the fragment joining of [10]. Details regarding this can be found in the full version of the paper [11].

In the sequel we assume that we have such a set, *Centers*.

Recall that in H, every super-vertex has in-degree exactly 1. For a super-vertex S (not necessarily a center), we define $\text{pred}(S)$ to be the first center we reach by starting from S and traversing backwards along reverse H edges. (Note that a super-vertex can be its own predecessor, if it is a center and is part of a directed cycle in H that includes no other centers.)

The *center graph* is the graph induced by pred:

▶ **Definition 5** (The center graph, H^*). *The* center graph *induced by H and Centers, denoted H^*, is given by $H^* = (\mathit{Centers}, \{(\text{pred}(c), c) \mid c \in \mathit{Centers}\})$.*

For an active component H_i, let H_i^* be the subgraph of H^* induced by the centers $\mathit{Centers} \cap H_i$ selected from H_i. Note the following properties: (1) Like H, the center graph H^* also has in-degree 1, except for the root (always selected as a center), which has no incoming edges; (2) If H_i includes a center, then H_i^* is a weakly-connected component of H^*; (3) Whenever $C(H_i)$ includes at least one center, H_i^* contains a non-empty cycle $C(H_i^*)$ (possibly one center with a self-loop), whose vertices are the centers from $C(H_i)$.

Finding $c(H_i)$. After setting up the centers and the center graph, to find some super-vertex from the cycle $C(H_i)$, we divide into three cases, depending on whether the active component H_i and its cycle $C(H_i)$ include more than $\sqrt{n}$ physical nodes or not.

1. H_i is a "small component", including at most $\sqrt{n}$ physical nodes: then in particular, the physical size of the cycle $C(H_i)$ does not exceed $\sqrt{n}$. We can find $C(H_i)$ by having each super-vertex start a forward-BFS along the edges of H for $O(\sqrt{n})$ rounds, and

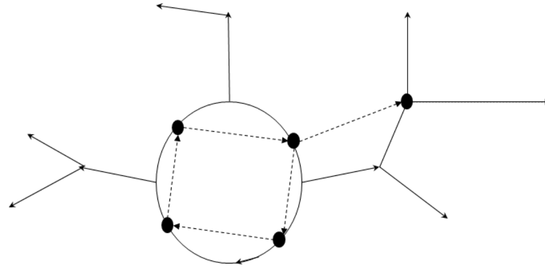


Figure 3 The center graph, overlayed on the super-vertex graph. Bold vertices represent centers, and dashed arrows represent the edges of the center graph.

always propagating the ID of the smallest super-vertex heard so far; after $O(\sqrt{n})$ rounds, some super-vertex receives back its own ID, and this super-vertex then becomes $c(H_i)$. We inform all nodes of H_i by propagating the ID of $c(H_i)$ for $O(\sqrt{n})$ rounds. This is handled by procedure `FindSmallCycles` (see details in full version [11]).

II. $C(H_i)$ is "small" (at most $\sqrt{n}$ physical nodes), but H_i is "large" (more than $\sqrt{n}$ nodes): in this case, procedure `FindSmallCycles` still selects some super-vertex $c(H_i) \in C(H_i)$ just as above. However, we cannot afford to disseminate the ID of $c(H_i)$ throughout H_i by broadcasting it, because H_i is too large. Instead, we add $c(H_i)$ to the center set *Centers*, and handle its dissemination below.

III. $C(H_i)$ is "large": then $C(H_i)$ includes at least one center, and we can identify $C(H_i)$ by examining the center graph H^* and looking for the corresponding cycle there.

In cases (II) and (III), after `FindSmallCycles` is called, $C(H_i)$ includes at least one center: either it was there before, or if the cycle was too small, we added some center in `FindSmallCycles`. Therefore, the component H_i^* that corresponds to H_i in the center graph contains a cycle $C(H_i^*)$.

After calling `FindSmallCycles`, every super-vertex S learns the identity of $\text{pred}(S)$. Because every H-path of physical size at least $\sqrt{n}$ includes a center, for each super-vertex S (not necessarily a center), the physical distance from the entry vertex of $\text{pred}(S)$ to some physical vertex in S is at most $\sqrt{n}$. Thus, the super-vertex $\text{pred}(S)$ can "tell S" that it is its predecessor by doing a forward BFS for $\sqrt{n}$ rounds (we omit the details here).

The center graph has $O(\sqrt{n})$ edges: its in-degree is 1, and even after adding some centers in step II, we still have $O(\sqrt{n})$ centers, because a center is only added for active components of physical size $> \sqrt{n}$. Thus, we can afford to disseminate all edges of H^* throughout the network, in $O(\sqrt{n} + D)$ rounds.

Finally, each physical node v locally examines the graph H^*, and constructs the weakly connected components of H^*. It associates itself with the correct component H_i^* by choosing the component of H^* that contains the center $\text{pred}(sId(v))$, that is, the predecessor of its own super-vertex. If H_i^* includes the root, then v sets the root's ID as its active component ID. Otherwise, node v finds $C(H_i^*)$, selects the center with the smallest id $c \in C(H_i^*)$, and sets $cId(v) = c$.

6 DMST vs. (s, t)-Shortest Path

We have shown that the DMST problem is no harder than single-source shortest path. In this section we adapt a reduction of Chechick [4] from the sequential setting to CONGEST, showing that distributed DMST is at least as hard as (s, t)-shortest path, where we are given

two vertices s, t and must find the shortest directed path from s to t. The reduction holds for all three models we consider in this paper (assuming we work with strongly-connected graphs): it simply modifies the graph on which we want to solve (s, t)-SP, so that any DMST on the graph will reveal the shortest path from s to t. We take care that the modified graph can be *simulated* by the original graph without much additional communication.

Given a graph $G = (V, E)$, we define a graph G' as follows (see Fig. 4): G' contains all vertices and edges of G, and in addition, for each vertex $v \in V$, we add a "shadow vertex" v', with a zero-weight edge (v', v). For each original edge (u, v) we add a "shadow edge" (u', v'), again with weight zero. Finally, we add the zero-weight edge (t, t') (where t is the target node).

Observe that all the edges we added to G' are either shadow edges or edges incoming into vertices of G, except for the edge (t, t'), which is outgoing from t. Therefore, in G', we did not create any path from s to t that was not already in G.

▶ **Lemma 6.** *The weight of the DMST of G' rooted at s is the weight of the (s, t)-shortest path in G.*

Proof. Let W' be the weight of the DMST of G' rooted at s, and let d be the weight of the shortest path from s to t in G. It is easy to see that $W' \geq d$, because the DMST must contain *some* path from s to t, and in G' we did not create any path from s to t that was not already in G.

To show that $W' \leq d$, consider the following DMST: take a shortest path π from s to t in G, and add all its edges to the DMST. In addition, take edge (t, t'), and some arbitrary directed spanning tree of the shadow vertices, comprising only shadow edges and oriented outwards from the root t'. (Such a spanning tree exists, because we can take a directed spanning tree of G rooted at t and "copy it" onto the shadow edges.) Finally, for each $v \in V$ that is not on π, add the edge (v', v). The resulting tree is spanning and oriented outwards from s, and its weight is exactly d, because other than the edges of π, it uses only zero-weight edges. ◀

▶ **Theorem 7.** *The asymptotic round complexity of DMST in* CONGEST *is at least that of (s, t)-shortest path.*

Proof. Given a DMST algorithm $\mathcal{A}$ and a graph G, we can solve (s, t)-shortest path on G by constructing G' and simulating the execution of $\mathcal{A}$ on G'. Each vertex v of G simulates itself and its shadow vertex v'. To simulate one round of $\mathcal{A}$ on G', each vertex sends to its neighbors the messages that it would send under $\mathcal{A}$ on its own edges, and also the messages its shadow vertex would send on *its* edges under $\mathcal{A}$. This increases the communication by only a constant factor. ◀

7 Finding the Directed Minimum-Weight Spanning Tree

In this section we describe how to find the edges of the DMST, after contracting the entire graph into one component. This is an adaptation of the unpacking procedore from Lovasz' DMST algorithm [24], implemented in CONGEST. Again, some technical details are omitted here.

Recall that when we performed contractions, we looked for an edge that minimizes the *entering distance* into H_i,

$$\beta(u, v) := w(u, v) + \mathrm{dist}_{H_i}(v, C(H_i)),$$

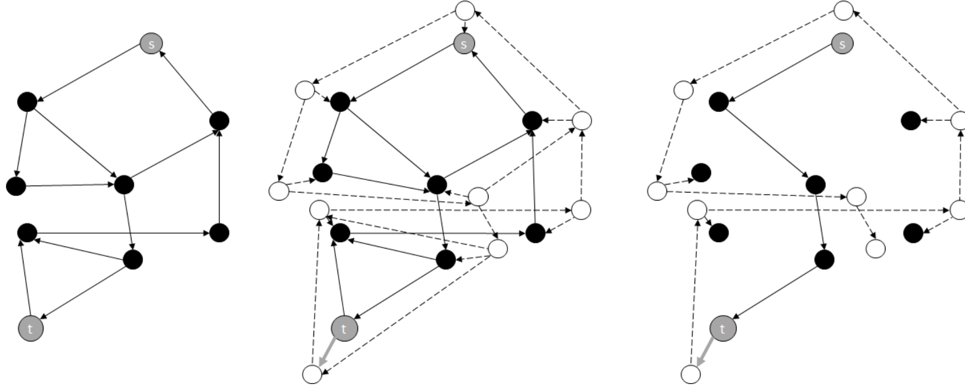


Figure 4 Local reduction of (s,t)-shortest path to DMST. The shadow nodes are shown in white. The rightmost figure shows the DMST.

and we denoted this minimum distance by

$$\beta_i = \min_{(u,v)\in \mathrm{In}(H_i)} \beta(u,v).$$

For an active component H_i, let $G_{\triangleright i}$ denote the contracted graph in which, starting from G, we contracted the cycle $C(H_i)$, together with all vertices inside H_i that have distance up to β_i from $C(H_i)$ (or rather, from the active component ID, $c(H_i)$), into one super-vertex. We now describe how to "undo" the contraction, so that we can unpack $G_{\triangleright i}$ back into G and add the correct edges to the DMST.

Unpacking a super-vertex. Consider a graph R with a set H of active edges, and let T' be a DMST of the contracted graph $R' = R_{\triangleright i}$. Let w, w' be the weight functions of R, R' respectively. Let $s = \bigcup U_i(\beta_i)$ be the new super-vertex in R' formed by merging together all vertices with distance at most β_i from $C(H_i)$ in R.

We define an *unpacking operation* that constructs from T' a new tree, denoted $T'_{i\triangleleft}$, for the original graph R, as follows:

- The new tree agrees with T' on all edges that are not adjacent to s: for any edge $e = (s_1, s_2)$ where $s_1, s_2 \neq s$ we have $(s_1, s_2) \in T'_{i\triangleleft}$ if $(s_1, s_2) \in T'$.
- Let (u, s) be the edge in T' that is incoming into s (there must be such an edge, since T' is spanning). Let $v^* \in s$ be the vertex that minimizes $\beta(u, v)$ among all incoming edges into s. We add to T the edge (u, v^*) and the lightest-weight path π from v to $C(H_i)$ in $R(H_i)$.
- If T' contains an outgoing edge (s, x) from s, each such edge is again replaced by an edge (y, x), where $y \in s$, that has $w(y, x) = \min_{y'\in s} w(y', x)$.
- Finally, we add to T the edges $H_i \setminus \mathrm{dest}(\pi)$, where $\mathrm{dest}(\pi)$ is the set of edges in R whose destinations are nodes on π.

▶ **Lemma 8** (Variant of [24], Claim 2). *If T' is a DMST of $R' = R_{\triangleright i}$, then $T'_{i\triangleleft}$ is a DMST of R.*

Unpacking the DMST. Now we describe how we "unpack" the entire DMST, starting from the final state of the algorithm where the graph has been contracted until only the root active component remains. The algorithm we describe here is run by the physical vertices of G, and

it runs in $\log(n)$ iterations, indexed downwards, $\log n, \dots, 1$, where the j-th iteration unpacks the super-vertices created at step j. Let $\left\{H_1^t, \dots, H_{k_i}^t\right\}$ be the set of active components in iteration i.

We may assume w.l.o.g. that in an SSSP algorithm for directed graph with non-negative integer weights, each node also outputs a parent in a SSSP tree (e.g [12]). We require the nodes to store these edges; specifically, each node needs to remember, for each contraction, its edges in the reverse shortest-paths tree from $c(H^j)$ that was computed during the contraction.

Each super-vertex created during the current iteration is unpacked in parallel, and the edges taken into the DMST are the edges described above. When choosing which of their edges to add, the only computation nodes cannot perform locally is to find the shortest-path edges from v^* into $c(H(v^*))$. We handle this using centers, just as we did in Section 5.2: if the path is short, we can find it by doing a short BFS; and if the path is long, it will contain at least one vertex, and we can use the center graph to have the vertices of the path learn that they are on the path and add edges accordingly.

References

1. Aaron Bernstein and Danupon Nanongkai. Distributed Exact Weighted All-Pairs Shortest Paths in Near-Linear Time. In *Proceedings of the 51th Annual ACM SIGACT Symposium on Theory of Computing*, STOC '19, 2019.
2. F. Bock. An algorithm to construct a minimum directed spanning tree in a directed network. In *Developments in Operations Research*, pages 29–44, 1971.
3. Keren Censor-Hillel, Petteri Kaski, Janne H. Korhonen, Christoph Lenzen, Ami Paz, and Jukka Suomela. Algebraic Methods in the Congested Clique. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015*, pages 143–152, 2015.
4. S. Chechik. Private Communication.
5. Mohit Daga, Monika Henzinger, Danupon Nanongkai, and Thatchaphol Saranurak. Distributed Edge Connectivity in Sublinear Time. In *Proceedings of the 51th Annual ACM SIGACT Symposium on Theory of Computing*, STOC '19, 2019.
6. Tarjan R. E. Finding optimum branchings. *Networks*, 7(1):25–35, 1965.
7. Jack Edmonds. Optimum branchings. *Journal of Research of the National Bureau of Standards*, 71B(4):233–240, 1967.
8. M. Elkin. An Unconditional Lower Bound on the Time-Approximation Trade-off for the Distributed Minimum Spanning Tree Problem. *SIAM Journal on Computing*, 36(2):433–456, 2006.
9. Michael Elkin. A faster distributed protocol for constructing a minimum spanning tree. *J. Comput. Syst. Sci.*, 72(8):1282–1308, 2006.
10. Michael Elkin. A Simple Deterministic Distributed MST Algorithm, with Near-Optimal Time and Message Complexities. In *Proceedings of the ACM Symposium on Principles of Distributed Computing, (PODC)*, pages 157–163, 2017.
11. Orr Fischer and Rotem Oshman. A Distributed Algorithm for Directed Minimum-Weight Spanning Tree. `https://www.cs.tau.ac.il/~roshman/papers/DISC19_FO.pdf`.
12. Sebastian Forster and Danupon Nanongkai. A Faster Distributed Single-Source Shortest Paths Algorithm. In *59th IEEE Annual Symposium on Foundations of Computer Science, (FOCS)*, pages 686–697, 2018.
13. H N Gabow, Z Galil, T Spencer, and R E Tarjan. Efficient Algorithms for Finding Minimum Spanning Trees in Undirected and Directed Graphs. *Combinatorica*, 6(2):109–122, January 1986.
14. J. A. Garay, S. Kutten, and D. Peleg. A sub-linear time distributed algorithm for minimum-weight spanning trees. In *Proceedings of 1993 IEEE 34th Annual Foundations of Computer Science*, pages 659–668, 1993.

15 Mohsen Ghaffari and Bernhard Haeupler. Distributed Algorithms for Planar Networks II: Low-Congestion Shortcuts, MST, and Min-Cut. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 202–219, 2016.

16 Mohsen Ghaffari and Jason Li. Improved Distributed Algorithms for Exact Shortest Paths. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2018, pages 431–444, 2018.

17 Mohsen Ghaffari and Merav Parter. MST in log-star rounds of congested clique. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC*, pages 19–28, 2016.

18 Bernhard Haeupler, Taisuke Izumi, and Goran Zuzic. Low-Congestion Shortcuts without Embedding. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing (PODC) 2016*, pages 451–460, 2016.

19 Bernhard Haeupler and Jason Li. Faster Distributed Shortest Path Approximations via Shortcuts. In *32nd International Symposium on Distributed Computing, (DISC) 2018*, pages 33:1–33:14, 2018.

20 Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. A Deterministic Almost-tight Distributed Algorithm for Approximating Single-source Shortest Paths. In *Proceedings of the Forty-eighth Annual ACM Symposium on Theory of Computing*, STOC '16, pages 489–498, 2016.

21 P. Humblet. A Distributed Algorithm for Minimum Weight Directed Spanning Trees. *IEEE Transactions on Communications*, 31(6):756–762, 1983.

22 Tomasz Jurdzinski and Krzysztof Nowicki. MST in $O(1)$ rounds of congested clique. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018*, pages 2620–2632, 2018.

23 Zvi Lotker, Boaz Patt-Shamir, Elan Pavlov, and David Peleg. Minimum-Weight Spanning Tree Construction in $O(\log \log n)$ Communication Rounds. *SIAM J. Comput.*, 35(1):120–131, 2005.

24 L. Lovasz. Computing ears and branchings in parallel. In *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)(FOCS)*, volume 00, pages 464–467, 1985.

25 Danupon Nanongkai. Distributed Approximation Algorithms for Weighted Shortest Paths. In *Proceedings of the Forty-sixth Annual ACM Symposium on Theory of Computing*, STOC '14, pages 565–573, 2014.

26 Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. A Time- and Message-optimal Distributed Algorithm for Minimum Spanning Trees. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2017, pages 743–756, 2017.

27 D. Peleg and V. Rubinovich. A near-tight lower bound on the time complexity of distributed MST construction. In *40th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 253–261, 1999.

28 F. Maffioli P.M. Camerini, L. Fratta. A note on finding optimum branchings. *Networks*, 7:309—-312, 1979.

29 F. Maffioli P.M. Camerini, L. Fratta. The k best spanning arborescences of a network. *Networks*, 10(2):91—-109, 1980.

30 Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. Distributed Verification and Hardness of Distributed Approximation. *SIAM J. Comput.*, 41(5):1235–1265, 2012.

31 Ramakrishna Thurimella. Sub-linear Distributed Algorithms for Sparse Certificates and Biconnected Components. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, 1995.

32 T.H. Liu. Y.J. Chu. On the shortest arborescence of a directed graph. *Sci. Sinica*, 14(2):1396–1400, 1965.

Stable Memoryless Queuing under Contention

Paweł Garncarek
Institute of Computer Science, University of Wroclaw, Poland
pgarn@cs.uni.wroc.pl

Tomasz Jurdziński
Institute of Computer Science, University of Wroclaw, Poland
tju@cs.uni.wroc.pl

Dariusz R. Kowalski
School of Computer and Cyber Sciences, Augusta University, Augusta, USA
SWPS University of Social Sciences and Humanities, Warsaw, Poland
dkowalski@augusta.edu

Abstract

In this work we study stability of local memoryless packet scheduling policies in a distributed system of n nodes/queues under contention. The local policies at nodes may only access their current local queues, and have no other feedback from the underlying distributed system. Moreover, their memory is limited to some basic parameters. The packets arrive at queues according to arrival patterns controlled by an adversary restricted only by injection rate ρ and burstiness b, or driven by a stochastic process; the former model analyzes worst-case stability while the latter – average case. We assume that the underlying distributed system is a classic shared channel, in which no two packets could be successfully scheduled (and removed from queues) at the same time. We show that there is a local memoryless scheduling policy which is both adversarially and stochastically stable for injection rates $\Omega(1/\log n)$. Another algorithm achieves even higher – constant – stable injection rate, but only for a bounded range of burstiness. The first algorithm is utilizing properties of interleaved ultra-selectors, for which we prove stronger properties than known so far, while the second one is based on entirely new concept of selector with thresholds, unlike previously considered binary selectors/codes in the literature.

Note that popular Backoff algorithms, some of which achieve stability for constant (stochastic) injection rates [18], use memory to record current state (e.g., the number of unsuccessful transmissions or the result of random sampling in each window) as well as randomization and feedback from the channel; unlike solutions in this work, which are memoryless and do not rely on randomization or channel feedback (thus, could be used independently from the link layer protocols).

2012 ACM Subject Classification Networks → Packet scheduling; Theory of computation → Online algorithms; Theory of computation → Distributed algorithms

Keywords and phrases packet scheduling, online algorithms, adversarial injections, stochastic injections, stability, memoryless algorithms

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.17

Funding This work was funded by Polish National Science Center grant 2017/25/B/ST6/02010.

1 Introduction

Recently, due to rapidly growing number of devices and popularity of distributed protocols, the impact of congestion on stability of queuing processes has become an important practical and research topic. They are everywhere, often dependent on each other and competing for the same resources (in this paper modeled as a shared channel with contention). Queues are governed by scheduling algorithms, often run locally in a distributed way. The desired property of the whole system of queues is stability – understood as existing of an upper bound on the numbers of packets queued at devices at any time. In this work, we study

© Paweł Garncarek, Tomasz Jurdziński, and Dariusz R. Kowalski;
licensed under Creative Commons License CC-BY

33rd International Symposium on Distributed Computing (DISC 2019).
Editor: Jukka Suomela; Article No. 17; pp. 17:1–17:16

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

stability of local memoryless packet scheduling policies in the process of dynamic distributed broadcasting on a shared channel. A shared channel, also called a multiple access channel, is a broadcast network with instantaneous delivery of transmitted messages to every device (also called a node or a station) in the system and a possibility of conflict for access to the transmitting medium. A message sent via a channel by a station is received successfully by all the stations when its transmission has not overlapped with transmissions by other stations. In the queue system in contention, the channel represents the contention and is interpreted as a rule that nothing happens to the queues if at least two of them schedule a stored packet/job to be transmitted/executed at the same time.

The traditional approach to modeling queuing process on a shared channel was through dynamic broadcasting problem and its corresponding queue system, but it could be easily generalized to arbitrary queues with jobs or other types of elements. It has assumed continuous packet injection subject to stochastic constraints (typically, Poisson arrival rates). Recent papers, following the adversarial queuing approach for store-and-forward packet networks, studied stability of the system of queues on a shared channel in adversarial settings. An adversary is determined by two parameters: injection rate ρ, which is the average number of injected packets, and burstiness b, which is the maximum number of packets that may be injected in a round.

We focus on memoryless schedulers, showing that, although very restricted, they are quite powerful in the sense that some of them could guarantee stability of the system for high injection rates. Memoryless schedulers are local policies at nodes, which may only access their current local queues, and have no other feedback from the underlying distributed system. The assumption of limited feedback assures that designed policies are applicable in broad spectrum of scenarios and systems; in particular they could adjust to multi-layer stacks of protocols by assuring independence of a scheduling policy from the actual feedback from the lower layers. Moreover, internal memory of nodes is limited to some basic parameters and $O(\log n)$ control bits per each packet stored (necessary to keep e.g., basic packet informations such as destination), where n is the number of queues in the system. This requirement addressess the need of optimization of resources in contemporary and emerging scenarios, e.g., networks of computationally limited wireless entities run on batteries in IoT applications.

1.1 Our results

This paper studies stability of deterministic local memoryless packet schedulers in the context of distributed broadcasting on a shared channel (as mentioned earlier, it could be generalized to other types of queues with contention). The stability is studied in adversarial setting (corresponding to worst case system behavior), defined in terms of global injection rate ρ and burstiness b, and stochastic setting (corresponding to average case system behavior), in which injections at each node follow the Bernoulli process. We show that there is a local memoryless scheduling policy with relatively small memory (i.e., one bit per packet, indicating whether the packet is old or relatively new), which is stable, both adversarially and stochastically, for injection rates $\Omega(1/\log n)$ on a shared channel. The scheduler is based on interwleaving ultra-selectors defined in [8]. Independently, we show better existential results for ultra-selectors. The second scheduler is based on a different type of selector with thresholds (unlike the previously used binary selectors/codes in the literature), and achieves stability for constant injection rates with bounded values of burstiness. Comparison of our results with the most relevant other recent results is given in Table 1.

1.2 Previous and related work

There is a long history of research on dynamic broadcast on multiple access channels. Early work includes developing and studying properties of protocols like Aloha [1] and binary exponential backoff [24]. Recent study on this topic has focused on scenarios when packets were injected subject to statistical constraints. Stochastic stability has been the basic quality criterion to achieve, understood in the sense that the input and output rates were equal. See the paper by Gallager [13] for an overview of early research; recent work includes the papers of Goldberg et al. [15], Goldberg et al. [16], Håstad et al. [18], Raghavan and Upfal [25], Bender et al. [5] and [22].

Adversarial queuing was introduced by Borodin et al. [7] as a framework to study stability of routing protocols in (point-to-point connected) networks under worst-case traffic scenarios modeled by adversaries. Independently, Andrews et al. [4] defined a greedy protocol to be universally stable when it was stable in all networks for any injection rate $\rho < 1$. This line of research for wired networks has been intensively pursued for many models and protocols.

Chlebus at al. [10] were the first who studied adversarial queuing on a shared channel. They however, similarly to all the follow up work (cf., [3, 2, 9]), assumed that schedulers are embedded into the channel, in the sense that they can receive channel feedback or even attach and read additional information bits. Several results were obtained and new protocols designed and analyzed using this model, however they were derived for stronger schedulers or for restartable schedulers (so called acknowledgment-based) which are incomparable to local memoryless class. Recently, Garncarek et al. [14] investigated adversarial stability and other properties of deterministic local packet schedulers. They considered two classes of local schedulers: adaptive and non adaptive. The former allows stations to monitor and store some digest of the local queue history (especially its size), which is much more powerful than memoryless schedulers considered in this work, while the latter allows the policy only to check whether the current local queue is empty or not, which in turn is a type of memoryless policy. They showed that there is a local adaptive scheduling policy with relatively small memory, which is universally stable on a shared channel, that is, it has bounded queues for any $\rho < 1$ and $b \geq 0$. On the other hand, they proved that memoryless policies with only information about non-emptiness of their queues could reach the maximal stable injection rate of $O(1/\log n)$. They also showed a local non-adaptive policy, which is stable for slightly smaller injection rate $c/\log^2 n$, for some constant $c > 0$. In this context, general memoryless local policies considered in this work could be seen as in-between of adaptive and no-adaptive local schedulers considered in [14].

A simplified version of broadcasting on a shared channel with static input, in which some k stations hold packets at the beginning of an execution, was also widely investigated. The goal is to transmit at least one of them (*selection problem*) or all of them (*k-broadcast problem*), in both cases minimizing time complexity. The selection problem was studied in particular by Kushilevitz and Mansour [23] and Willard [26]. The k-broadcast problem was studied by Greenberg and Winograd [17], Komlós and Greenberg [21], and Kowalski [22]. A related leader election problem was studied by Jurdziński et al. [19] for channels without collision detection.

Deterministic solutions for the mutual exclusion and consensus problems on multiple-access channels when the adversary wakes up stations in arbitrary rounds were studied by Czyżowicz et al. [11]. A randomized counterpart of their research was delivered by Bieńkowski et al. [6].

Table 1 Comparison of our algorithms (WBA and QSA) with the previously known results under adversarial packet injections. Row with ρ describes the highest injection rate for which an algorithm is stable. Control messages can be separate packets (that need an additional successful transmission), piggybacked on packets (sent in the same round as successful transmission of any packet) or neither. Local memory is the number of bits to store information about history of events that each node can remember. Queue access denotes whether a node has access to the size of its queue or only knows if its queue is empty.

algorithm	oblivious [14]	WBA	QSA	adaptive [14]	MBTF [9]
ρ	$O(1/\log^2 n)$	$O(1/\log n)$	$O(1)$	< 1	≤ 1
b	any	any	bounded	any	any
control messages	no	no	no	separate	piggybacked
local memory	0	$O(1)$	0	large	$O(n \log n)$
queue access	emptiness	size	size	size	size

2 Model

We follow the description of local scheduling model under contention from [14], enhanced by additional aspects of memoryless policies. There are n stations attached to a shared channel. The stations have distinct names in $[n] = \{0, 1, \ldots, n\}$. Each station v knows n and its name $v \in [n]$.

2.1 Shared channel

A shared channel, also called a multiple access channel, models environments in which distributed nodes compete for access to the shared communication and distribution channel, and in case of contention, no contender wins the access. We assume that the channel operates synchronously. Every station (also called a node or queue) connected to it has its clock and the clock cycles are all of exactly the same length and synchronized. An execution of a protocol is partitioned into rounds – it takes precisely one round to transmit a message. We assume that stations have access to a global clock, meaning that all the local clocks at stations read the same round numbers.

Every station occasionally receives packets to broadcast. Packets are stored in a local queue. Stations use local scheduling algorithms to decide whether to schedule a packet from the queue for transmission in the current round or not. When exactly one station schedules a packet for transmission in a round, then the message containing this packet is successfully delivered and the packet disappears from the queue. We assume that local schedulers do not receive any feedback from the channel – they could only make their decisions based on examining local queue. When at least two stations transmit simultaneously in a round then *conflict for access* or *collision* occurs in the round and none of the transmitted packet is successful (i.e., all remain in their local queues). We consider syntactically weaker, but as we will show still powerful, *memoryless* local policies, in which stations may only access their current local queues and have no other feedback from the underlying distributed system. Moreover, the size of their internal writeable memory is bounded by some basic parameters and $O(\log n)$ control bits per each packet stored, storing e.g., packet destination or estimate of arrival time – see Table 1 for details. W.l.o.g. we assume that the queue at a station operates in the first-in-first-out (FIFO) fashion, as we are only interested in stability (i.e., bounded queue sizes).

2.2 Packet injections

Packets are injected into stations in a dynamic fashion in the course of an execution of a broadcasting protocol. We consider two packet injections – adversarial, corresponding to worst case executions, and stochastic, corresponding to average case executions.

Consider packet injection modeled by an adversary. Adversaries are specified by constraints on the maximum rate of injection ρ and by the burstiness of traffic b. An adversary generates a number of packets in each round and for each packet assigns a station to inject the packet in this round. The number of packets an adversary can inject into stations in one round is called the *burstiness* of the adversary. The adversary of type (ρ, b) can inject at most $\rho t + b$ packets to stations, in total, during any time interval of t rounds. This type of adversary is typically called *leaky bucket*.

We consider stochastic packet injections that are i.i.d. across rounds and independent across nodes. The probability that a (exactly one) packet is injected in a round t into a node v is ρ_v. Thus, each node receives packets in each round according to Bernoulli process. The value $\rho = \sum_v \rho_v$ is called the injection rate. We will only consider injections such that $\rho \leq 1$, since otherwise the system will obviously accumulate infinitely many packets over time.

2.3 Quality of service

We say that a local scheduler is *stable* for injection rate ρ if in any execution of the scheduler against a (ρ, b)-adversary, for any b, queues are bounded at all times. *Stochastic stability* means that the Markov Chain associated with the execution (random, due to stochastic injections) has a positive recurrent set of states with low queues, i.e., starting from any queue size, the expected time until the system contains few packets is bounded.

3 Generic queuing

In this section we present a queuing algorithm in the window-based model, which works for arbitrary burstiness. Thanks to that, our algorithm is stable for both adversarial (worst-case) and stochastic (average) packet injections. It is based on ultra-selectors – a combinatorial tool introduced in [8].

▶ **Definition 1** ([8])**.** *For a given $1 \leq a \leq n$ and $0 < \varepsilon \leq 1$ a (n, a, ε)-US (ultra-selector) of length m is a family of sets $S_1, \ldots, S_m$ such that for any set $A \subseteq [n]$ of size at most a and more than $a/2$, at least a ε fraction of the sets in the family intersect A on a single element: the size of $\{i \in [m] \mid |S_i \cap A| = 1\}$ is at least εm.*

▶ **Lemma 2** ([8])**.** *For each $\varepsilon < 1/32$ and sufficiently large n, there is an integer $c > 0$ such that for each $1 \leq a \leq n$, there exists an (n, a, ϵ)-US of length at most $m = c \cdot a \log(2n/a)$.*

We improve the existential result from [8] by extending the range $\varepsilon \in (0, 1/32)$ to $\varepsilon < (0, 1/(2e))$.

▶ **Lemma 3.** *For any $\epsilon < 1/(2e)$ there is an integer $c > 0$ such that for $1 \leq a \leq n$, there exists an (n, a, ϵ)-US of length at most $m = c \cdot a \log(2n/a)$.*

Proof. Observe that for $n/2 < a \leq n$ it is enough to take the family of all singletons of set $[n]$, as for any admissible set the number of its singleton intersections with the sets in the family is bigger than $a/2 > n/4$, which gives $\epsilon \geq 1/4 > 1/(2e)$ even better than the one claimed in the lemma.

It remains to consider $1 \leq a \leq n/2$. We show the existence by using the probabilistic method. Let each set of the (n, a, ϵ)-US be selected independently at random as follows, and the number of sets be $c \cdot a \log(2n/a)$, for some constant $c > 0$, depended on ϵ, to be specified later. For each set and integer $v \in [n]$, element v belongs to the set with probability $1/a$, independently on other elements and sets.

Consider an arbitrary set $A \subseteq [n]$ such that $a/2 < |A| \leq a$. For any set T of the randomly created (n, a, ϵ)-US, the probability that $|T \cap A| = 1$ is

$$|A| \cdot \frac{1}{a} \cdot \left(1 - \frac{1}{a}\right)^{|A|-1} \geq \frac{1}{2} \cdot \left(1 - \frac{1}{a}\right)^{a-1} \geq \frac{1}{2e} .$$

Thus the expected number of sets T in the (n, a, ϵ)-US such that $|T \cap A| = 1$ is at least $1/(2e) \cdot c \cdot a \log(2k/a)$. By Chernoff bounds, the probability that the number of such sets is smaller than $\epsilon \cdot c \cdot a \log(2n/a)$ is less than

$$\exp\left(-1/(2e) \cdot c \cdot a \log(2n/a) \cdot (1/(2e) - \epsilon)^2 \cdot 1/2\right) = \exp\left(-c \cdot a \log(2n/a) \cdot \frac{1 - 2e\epsilon}{8e^2}\right) . \quad (1)$$

The number of all possible sets $A \subseteq [n]$ satisfying $a/2 < |A| \leq a$ is

$$\sum_{i=a/2+1}^{a} \binom{n}{i} \leq \frac{a}{2} \cdot \binom{n}{a} \leq \frac{a}{2} \cdot \left(\frac{ne}{a}\right)^a , \quad (2)$$

as $\binom{n}{x}$ are monotonically increasing with growing $x \leq a \leq n/2$. Therefore, the probability that there is an admissible set A having less than $\epsilon \cdot c \cdot a \log(2n/a)$ singleton intersections with the random (n, a, ϵ)-US is at most

$$\frac{a}{2} \cdot \left(\frac{ne}{a}\right)^a \cdot \exp\left(-c \cdot a \log(2n/a) \cdot \frac{1 - 2e\epsilon}{8e^2}\right) \leq$$

$$\leq \exp\left(\ln(a/2) + a \ln(ne/a) - c \cdot a \log(2n/a) \cdot \frac{1 - 2e\epsilon}{8e^2}\right) .$$

This in turn is smaller than 1 for sufficiently large constant c, depending on ϵ. By the probabilistic method, there is a (deterministic) (n, a, ϵ)-US of length $c \cdot a \log(2n/a)$. ◀

In Sections 3.1–3.3, we devise a scheduling algorithm in window-based model which uses ultra-selectors as a building block and show its stability against adversarial as well as stochastic injections.

3.1 Algorithm

Our algorithm WBA (Window-Based Algorithm) makes use of ultra-selectors $(n, 2^i, \varepsilon)$-US, with $i = 0, 1, \ldots, \log n - 1$ such that the length of $(n, 2^i, \varepsilon)$-US is at most $c \cdot 2^i \log(2n)$ for the constant c from Lemma 2. Let L_i denote the length of the used $(n, 2^i, \varepsilon)$-US. Let j be the smallest number such that $2^j \geq L_i$ for all i. The algorithm works in windows which consist of $L = 2^j \log n$ rounds (see Algorithm 1 on page 7 for a pseudocode for a window). A node is *active* in a window if the number of packets in its queue at the beginning of the considered window is larger or equal to $\varepsilon 2^j$. Windows are split into $\log n$ phases of length 2^j each. During the ith phase of a window $(n, 2^{\log n - i}, \varepsilon)$-US is "repeated" $r_i = \lfloor 2^j / L_{\log n - i} \rfloor$ times and the remainder of the phase is wasted, i.e., no nodes try to transmit. (Note that the wasted period of a phase is no longer than half of the phase.) The ith phase of a window

is determined by a chosen $(n, 2^{\log n - i}, \varepsilon)$-US of length $m \le 2^j$ in the following way. Let $S_1, \ldots, S_m$ be the chosen $(n, 2^{\log n - i}, \varepsilon)$-US. Then, S_k for each $k \in [m]$ is the set of *potential transmitters* for all rounds $k + lm$, such that $l \in [0, r_i]$. A node v transmits in the round t of the considered window if it is active in the current window, v belongs to the set of potential transmitters for round t and its queue is not empty.

Algorithm 1 Window(v). ▷ A window of Algorithm WBA

```
1: if |Q_v| ≥ ε2^j then                          ▷ Q_v: the queue of packets of v
2:     s_v ← active
3: else s_v ← non-active
4: for i = 1, ..., log n do                                          ▷ Phase i
5:     (S_1, ..., S_m) ← the chosen (n, 2^{log n − i}, ε)-US              ▷ m ≤ 2^j
6:     for t = 1, ..., 2^j do                                ▷ Round t of Phase i
7:         P_t ← S_{1+(t−1) mod m}
8:         if s_v=active and Q_v ≠ ∅ and v ∈ P_t then
9:             v transmits a message in round t of phase i
```

In the next two sections we will prove the following result.

▶ **Theorem 4.** *Algorithm WBA achieves stability against adaptive adversaries with injection rate $\rho \le \varepsilon/(2\log n)$ and any burstiness b, where ε is a fraction of successes of $(n, 2^i, \varepsilon)$-US used. Moreover, WBA achieves stochastic stability against stochastic arrivals with injection rate $\rho' < \varepsilon/(2\log n)$.*

3.2 Proof of part 1 of Theorem 4 – Adversarial analysis

Consider a window. We say that the ith phase of the window is *efficient* if the number of active nodes x in the current window satisfies the inequalities $2^{\log n - i - 1} \le x \le 2^{\log n - i}$, for $i = 0, 1, \ldots, \log n - 1$. We split further analysis into two scenarios.

Scenario 1. There is an efficient phase in the window.

If the ith phase is efficient, then the active nodes during this phase transmit without collisions in ε fraction of non-wasted rounds, i.e., they successfully transmit $\varepsilon 2^j/2$ times during the phase, unless some (at least one) active node u has not enough packets to do so. In the latter case, the node u transmitted all packets present in Q_u at the beginning of the current window. As u is active, the number of packets in Q_u at the beginning of the current window is at least $\varepsilon 2^j$. In either case there are at least $S = \varepsilon 2^j/2 \ge \rho L$ successful transmission since the start of the window by the active nodes.

Scenario 2. There is no efficient phase in the window.

In this case, there are no active nodes in the window and thus each node has less than $\varepsilon 2^j$ packets in its queue at the beginning of the window. As the adversary can inject at most $\rho 2^j \log n + b = \varepsilon 2^j/2 + b = S + b$ packets during the window, there are at most $\rho 2^j(n+1) + b$ packets in the system in each round of the window (in particular, at the end of the window).

Thus, one of the following two conditions is satisfied for each window W:

- Case 1: Window W satisfies Scenario 1. Consider a sequence of windows $W = W_1, W_2, \ldots, W_p$ such that W_{i+1} directly precedes W_i, Scenario 1 holds for the windows $W_2, \ldots, W_p$ and either W_p is the first window during an execution of the algorithm or the window W_{p+1} preceding W_p satisfies Scenario 2. Then, there are at most $(n+1)\rho L + b$

packets at the beginning of W_p (see Scenario 2). Moreover, at most $\rho p L + b$ packets are injected in the time period corresponding to the windows $W_p, W_{p-1}, \ldots, W_1$. On the other hand, at least ρL packets are successfully transmitted in each of the windows $W_1, \ldots, W_p$. Therefore, the number of packets in all queues at the end of W_1 is at most $(n+1)\rho L + 2b$.

- Case 2: Window W satisfies Scenario 2. As argued above in Scenario 2, there are at most $\rho 2^j(n+1) + b$ packets at the end of the window.

So the number of packets in the system is at most $P = (n+1)\rho L + 2b + \rho 2^j(n+1) + b$ at the end of each window. Therefore, there are at most $P + \rho L + b$ packets at each round of an execution of the algorithm, which proves its stability.

3.3 Proof of part 2 of Theorem 4 – Stochastic analysis

We will prove that our algorithm is stochastically stable. That is to say, we will show that the Markov Chain described by the queue states has a positive recurrent set of states with low queues, i.e., starting from any queue size, the expected time until the system contains few packets is finite.

Note that the expected number of packets injected into the system during a window of length $L = 2^j \log n$ equals $E(\sum_{i=1}^{L} X_i) = L \cdot E[X] = \rho 2^j \log n$.

In a window, we consider two scenarios: Scenario (1) – there is an efficient phase; Scenario (2) – there is no efficient phase.

In Scenario (2) there are no active nodes in the window and thus each node has less than $\varepsilon 2^j$ packets in its queue at the beginning of the window. As packet injections into each node are described by binomial distribution, each node may receive at most 1 packet per round. Therefore, during a window, each node can receive at most $L = 2^j \log n$ packets. So, at the end of the window, there are at most $n \cdot (\varepsilon 2^j + 2^j \log n)$ packets in the system.

In Scenario (1), as shown earlier (see Scenario (1) of the adversarial analysis), at least $s \geq \varepsilon 2^j/2$ packets are successfuly transmitted during the window. This means that the expected change in the total number of packets in the system is $\Delta \leq E(\sum_{i=1}^{L} X_i) - s \leq \rho \cdot 2^j \log n - \varepsilon 2^j/2$. For $\rho \leq (\varepsilon - \delta)/(2 \log n)$ with some constant $\delta > 0$, we get $\Delta \leq (\varepsilon - \delta)/(2 \log n) \cdot 2^j \log n - \varepsilon 2^j/2 = -\delta 2^j/2 < 0$.

We will use Foster-Lyapunov Theorem to prove that there will only be (in expectation) a finite number of windows of the type (2), before a window of the type (1) is reached.

Let Q_i denote a vector of queue sizes of all nodes at round i of the algorithm execution. Note that Q_{i+1} depends on the value of Q_i, the stochastic injections (which are i.i.d. across rounds) and algorithm's transmission (which are a function of Q_i). Therefore, (Q_i) is a time-homogeneous Markov chain.

▶ **Theorem 5** (Foster-Lyapunov Theorem (see Theorem 1 in [12])). *Consider a time-homogeneous Markov chain (X_i). Suppose that the drift $E[V(X_1) - V(X_0)|X_0 = x]$ of some function V in one step satisfies the following conditions, for some positive N_0, c, and H:*

$$E[V(X_1) - V(X_0)|X_0 = x] \leq -c \quad \text{if} \quad V(x) > N_0,$$

$$E[V(X_1) - V(X_0)|X_0 = x] \leq H < \infty \quad \text{if} \quad V(x) \leq N_0$$

Then the set $B = \{x : V(x) \leq N_0\}$ is positive recurrent.

Consider a function $V(Q)$ that assigns to a queue state Q the number of packets in the system. We will show that set $B = \{q : V(q) \leq n\varepsilon 2^j + nL\}$ is positive recurrent.

At the end of a window of type (1), the queue sizes are bounded by $n\varepsilon 2^j + nL$. This means that, starting from any queue sizes, the system will return in expected finite time to bounded queues, i.e., the algorithm is stochastically stable.

Now we will estimate the drift of function L at the ends of consecutive windows.

$$E[V(Q_1) - V(Q_2)|Q_1 = q \text{ yields a window of type (2) }]$$
$$= V(Q_1) - E[V(Q_1) + \Delta|Q_1 = q \text{ yields a window of type (2) }] \le -\delta/(2\log n) < 0$$

Therefore, according to the Foster-Lyapunov Theorem, after a finite (in expectation) number of windows, the system returns to set B of states, i.e., to a state that yields a window of type (1), and after that window, the queues are bounded by $n\varepsilon \cdot 2^j + nL$. Therefore, our algorithm is stochastically stable.

4 Queueing in the model without memory

In this section we consider the qsa model in which nodes do not have memory to write any information about history of an execution of an algorithm. That is, at each round t, a node i has only access to: the size of the network n, the value t of the global clock, its own ID i, the size $q_{i,t}$ of its queue of packets, and the upper bound on burstiness b. Thus importantly, the nodes can not store any information about history of computation and communication. In the remaining part of this section, we prove the following theorem.

▶ **Theorem 6.** *There exists a constant $\rho > 0$ such that for each $b, n > 0$, there exists a scheduling algorithm in qsa model which is stable against each adversary with injection rate ρ and burstiness b.*

4.1 Overview of the proof of Theorem 6

As before, our algorithm relies on an appropriate combinatorial structure determining behaviour of nodes in particular rounds of a window, where the number of rounds of the window corresponds to the size of the underlying combinatorial structure. Algorithm QSA uses $\log n$ different ultra-selectors in order to adjust to the unknown number of active nodes (i.e., the nodes with sufficiently many packets in their queues) in the current window . The fact that we need $\log n$ different selectors limits acceptable injection rate to $O(1/\log n)$. In order to achieve stability for injection rates $O(1)$, we build a new combinatorial structure which adjusts to the actual number of active nodes. However, we face here additional difficulty: the nodes do not have opportunity to store information about history, therefore "activity" can be determined merely on the current size of the queue of the node (not fixed for the whole window, as in Algorithm QSA). In order to overcome this additional difficulty and simultaneously improve acceptable injection rate to $O(1)$, the new combinatorial structure is not just a sequence of sets determining potential transmitters. Instead, each element (i.e., each round of the algorithm) of the structure is a vector of n thresholds $[M_1, \ldots, M_n]$. In our algorithm, the node $i \in \{1, \ldots, n\}$ is a potential transmitter in a round corresponding to $[M_1, \ldots, M_n]$ iff the number $|Q_i|$ of packets in the queue of i is at least M_i. Moreover, in order to prevent the adversary from malicious adjustment of sizes of queues (by injections) to the thresholds, we use thresholds which are multiplicities of some parameter depending on the burstiness b.

The general idea of the construction of the new combinatorial structure, called a *capacitated selector* is as follows. Assume that the sum of the sizes of queues $\sum_{i=1}^{n} |Q_i|$ is at most $S \gg b$. Then, there is at most 1 node with at least S packets, at most 2 nodes with at least $S/2$ packets each and in general there are at most 2^i nodes with at least $S/2^i$ packets each. On

the other hand, if the sum of the sizes of queues is at least $S/2$ and at most S, then there is at least one $i \in [\log n]$ such that the number of nodes with queue sizes $\geq S/2^i$ is in the range $[2^{i-2}, 2^i]$. (If the number of nodes with queue sizes $\geq S/2^i$ were greater than 2^i for any $i \in [\log n]$, then the sum of the queue sizes would be greater than S. If the number of nodes with queue sizes $\geq S/2^i$ were smaller than 2^{i-2} for all $i \in [\log n]$, then the sum of the queue sizes would be smaller than $S/2$.) Therefore, if only nodes with queue sizes $\geq S/2^i$ are potential transmitters, we can use $(n, 2^{i-2}, \varepsilon)$-US or $(n, 2^{i-1}, \varepsilon)$-US. As we do not know the particular i satisfying these conditions, a direct application of ultra-selectors would require to check all values of $i \in [\log n]$ which requires $\log n$ rounds and results in the injection rate $O(1/\log n)$, as in Theorem 4. Instead, we "compress" all these $\log n$ selectors using thresholds in the following way. We choose the threshold $[M_1, \ldots, M_n]$ determining possible transmitters such that $M_j = 2^i$ with probability $p_i = \frac{2^i}{cS}$ for some (fixed) constant c. This assignment assures that the expected number of nodes exceeding their thresholds is $\Theta(1)$. Using Probabilistic Method, we show that such adjustment of thresholds gives constant fraction of successful rounds with non-zero probability, for large enough sequence of rounds. This in turn guarantees that the number of all packets in the system decreases in a window, provided the number of packets was in the range $[S/2, S]$ at the beginning of that window.

The above ideas need further modifications to obtain an actual scheduling algorithm guaranteeing stability for constant injection rates. In particular, the fact that the sizes of queues can change both by successful transmissions and packet injections has to be taken into account in the the estimation of probability that a chosen set of thresholds guarantees sufficiently many successful rounds for **each** possible initial queue sizes and adversarial injections.

4.2 Proof of Theorem 6

Our algorithm QBA (Query-size Based Algorithm) divides time into windows of length L (to be fixed later). Behavior of each node in each window is determined by a fixed matrix M over natural numbers with n rows and L columns. The node i transmits a packet in round t of a window iff the number $q_{i,t}$ of packets in its queue at the beginning of that round is larger than or equal to $M_{i,j}$. For a fixed matrix M, QBA(M) denotes the instance of QBA algorithm which uses the matrix M in the above described way.

In order to adjust requirements sufficient for stability of an algorithm determined by such a matrix M, we characterize states of the network (i.e., sizes of queues) and adversarial injections by appropriate matrices. To this aim, we introduce the following combinatorial structure.

▶ **Definition 7.** *Given natural numbers n, b and $0 < \rho < 1$, a matrix M over positive natural numbers with n rows and L columns is a (n, ρ, b)* capacitated adversarial selector *((n, ρ, b)-cas) of size L iff there exists a natural number m, called the* load *of M, such that*

$$\sum_{i=1}^{n} q_{i,L} \leq m \tag{3}$$

for each sequence $q_{1,0}, \ldots, q_{n,0} \in \mathbb{N}$ such that $\sum_{i=1}^{n} q_{i,0} \leq m$ and each matrix $A \in \mathbb{N}^{n \times L}$ such that $\sum_{i=1}^{n} \sum_{t=1}^{L} A_{i,t} \leq \rho L + b$, where

$$q_{i,t} = \begin{cases} \max(0, q_{i,t-1} + A_{i,t} - 1) & \text{if } q_{i,t-1} + A_{i,t} \geq M_{i,t} \text{ and } q_{j,t-1} + A_{j,t} < M_{j,t} \\ & \text{for each } j \neq i \\ q_{i,t-1} + A_{i,t} & \text{otherwise} \end{cases}$$

for each $i \in [n]$ and $t \in [L]$.

Intuitively, the vector $q_{1,0},\ldots,q_{n,0}$ in the above definition corresponds to the sizes of queues at the beginning of a window of the algorithm QBA executed according to the matrix M. Then, $A_{i,t}$ is equal to the number of packets injected in the queue of the station i at round t of the window. Moreover, $q_{i,t}$ describes the number of packets in the node i after round t of the window. The assumption $\sum_{i=1}^{n}\sum_{t=1}^{L} A_{i,t} \leq \rho L + b$ corresponds to the restriction on a leaky bucket (ρ, b) adversary. The condition $q_{i,t-1} + A_{i,t} \geq M_{i,t}$ corresponds to node i transmitting at round t, while the condition $q_{j,t-1} + A_{j,t} < M_{j,t}$ for each $j \neq i$ corresponds to the situation that all nodes other than i are not transmitting in round t. Finally, the inequality $\sum_{i=1}^{n} q_{i,L} \leq m$ implies that the number of packets in all queues does not exceed m at the end of the window, provided the number of packets in queues at the beginning of the window, $\sum_{i=1}^{n} q_{i,0}$, is at most m as well. Below, we formalize this intuition.

▶ **Lemma 8.** *Assume that there exists $0 < \rho < 1$ such that, for each sufficiently large $n \in \mathbb{N}$ and each natural b, there exists a (n, ρ, b)-cas $M \in \mathbb{N}^{n,L}$ for some $L \in \mathbb{N}$, with load $m > \rho L + b$. Then, the algorithm QBA(M) is stable against a (ρ, b) leaky bucket adversary.*

Proof. Let $M \in \mathbb{N}^{n,L}$ be a (n, ρ, b)-cas with load $m > \rho L + b$. We prove that the overall number of packets in all queues is at most m over an execution of QBA(M) against a (ρ, b) leaky bucket adversary. The proof goes by induction with respect to the number of windows.

As discussed above, if the sizes of queues at the beginning of the window are equal $q_{1,0},\ldots,q_{n,0}$ and $A_{i,t}$ denotes the number of packets injected by the adversary at round t of the window in the queue of the node i, then $q_{i,t}$ (defined as in Def. 7) for $i \in [n]$ and $t \in [L]$ denotes the number of packets in the queue of i after round t, for each $i \in [n]$ and $t \in [L]$. There are no packets at the beginning of the first window of an execution of the algorithm. Thus the number of packets at the end of that window is at most $\rho L + b \leq m$, since the adversary can inject at most $\rho L + b$ packets in L rounds. For the inductive step assume that the overall number of packets at the beginning of the jth window of the execution is $\sum_{i=1}^{n} q_{i,0} \leq m$. Then, the overall number of packets at the end of the window is at most $\sum_{i=1}^{n} q_{i,L}$, where $q_{i,t}$ are determined as in Definition 7. As M is (n, ρ, b)-cas with load m, the final number of packets at the end of the considered window is at most $\sum_{i=1}^{n} q_{i,L} \leq m$. ◀

Given the above connection between capacitated adversarial selectors and QBA algorithm, it is sufficient to show that (n, ρ, b)-cas exists for a constant $\rho > 0$.

▶ **Lemma 9.** *There exists a constant $\rho > 0$ such that for each large enough $n \in \mathbb{N}$ and each $b \geq 0$, there exists a (n, ρ, b)-cas M of length $L = O(n \log n + b)$ with load $m = 8nL$.*

Observe that Th. 6 follows directly from Lemmas 8 and 9. Thus, it remains to prove Lem. 9.

4.3 Proof of Lemma 9

In order to emphasize connections with the algorithm QSA, the indices $i \in [n]$ are called nodes, and $t \in [L]$ are called rounds.

Using Probabilistic Method, we will show that for each n and b, there exist L, m and a matrix M which guarantees the properties stated in Definition 7. More specifically, we prove the lemma for each $\rho < 2^{-\frac{33}{8}}$, some $L = O(n \log n)$ such that $L \geq \rho L + b$ and load $m = \frac{1}{2} S_{\max} L$, where $S_{\max} = 16n$.

Consider any $q_{1,0},\ldots,q_{n,0}$ such that $\sum_{i=1}^{n} q_{i,0} \leq m$ and $A_{i,t}$ such that $\sum_{i=1}^{n}\sum_{t=1}^{L} A_{i,t} \leq \rho L + b \leq L$. Let us consider two cases:

Case 1: $\sum_{i=1}^{n} q_{i,0} \leq m/2$. Then, $\sum_{i=1}^{n} q_{i,L} \leq \sum_{i=1}^{n} q_{i,0} + \sum_{i=1}^{n}\sum_{t=1}^{L} A_{i,t} \leq \frac{1}{2}m + \rho L + b \leq \frac{1}{2}m + L < m$, and therefore the statement of the lemma holds.

Case 2: $m/2 \le \sum_{i=1}^{n} q_{i,0} \le m$. We consider Case 2 in the remaining part of the proof. Let (round) $t \in [L]$ be *successful* if $q_{i,t-1} + A_{i,t} \ge M_{i,t}$ and $q_{i,t-1} + A_{i,t} > 0$ for exactly one value of $i \in [n]$. Then

$$\sum_{i=1}^{n} q_{i,L} \le \sum_{i=1}^{n} q_{i,0} + \sum_{i=1}^{n}\sum_{i=1}^{L} A_{i,t} - |\{t \in [L] \,|\, t \text{ is successful}\}$$

Now, our goal is to show that there exists matrix M which guarantees that $\sum_{i=1}^{n}\sum_{i=1}^{L} A_{i,t} - |\{t \in [L] \,|\, t \text{ is successful}\}| \le 0$, i.e., $|\{t \in [L] \,|\, t \text{ is successful}\}| \ge \rho L + b$.

Consider a random matrix M with n rows and L columns such that

$$M_{i,t} = \begin{cases} L \cdot 2^k - t & \text{with probability } \dfrac{2^k}{c \cdot S_{max}} \text{ for } k \in [\log S_{\max}] \\ \infty & \text{with probability } 1 - \sum_{j=1}^{\log S_{\max}} \frac{2^j}{c \cdot S_{\max}}, \end{cases} \tag{4}$$

where $c = 4$. Our final goal is to show that such a matrix is (n, ρ, b)-cas of size L with load m with non-zero probability which implies that such a matrix exists.

Given the above description of the probabilistic choice of M, we introduce some auxiliary terminology and examine its properties. We say that (the node) i has $s_{i,t} = \lceil (q_{i,t} + t)/L \rceil$ *blocks* at round t. Below, we observe that the number of blocks in a node i can change (at most) twice in rounds $1, \dots, L$ and this prospective changes are just increments by one.

▶ **Proposition 10.** *Let $s_{i,0}, \dots, s_{i,L}$ be the number of blocks of the node i in rounds $1, \dots, L$. Then, $s_{i,0} \le s_{i,1} \le \dots \le s_{i,L} \le s_{i,0} + 2$, provided that $\rho L + b \le L$.*

Proof. Note that $q_{i,t+1} \ge q_{i,t} - 1$ and therefore $s_{i,t+1} = \lceil (q_{i,t+1} + (t+1))/L \rceil \ge \lceil (q_{i,t} - 1 + (t+1))/L \rceil = s_{i,t}$. On the other hand, $q_{i,t} \le q_{i,0} + \sum_{t'=1}^{t} A_{i,t'} \le q_{i,0} + \rho L + b \le q_{i,0} + L$ and therefore
$s_{i,t} \le \lceil (q_{i,0} + L + t)/L \rceil \le \lceil q_{i,0}/L \rceil + 2 = s_{i,0} + 2$. ◀

The following proposition shows that the number of all blocks is limited for all $t \in [L]$. Intuitively, it follows from the facts that the initial number of packets is at most $m = O(nL)$, the sum of "injections" is $\sum_{i=1}^{n}\sum_{t=1}^{L} A_{i,t} = O(L)$, while the size of a block is of the order of L.

▶ **Proposition 11.** *The overall number of blocks $S_t = \sum_{i=1}^{n} s_{i,t}$ in all nodes at round $t \in [L]$ is at least $S_{\max}/4$ and at most $S_{\max}$, provided that $\sum_{i=1}^{n} q_{i,0} \in [m/2, m]$, $m = \frac{1}{2} S_{\max} L$ where $S_{\max} = 16n$.*

Proof. At the beginning ($t = 0$), we have

$$\sum_{i=1}^{n} s_{i,0} = \sum_{i=1}^{n} \lceil q_{i,0}/L \rceil \le \sum_{i=1}^{n} (q_{i,0}/L + 1) \le \frac{1}{L} m + n = \frac{1}{2} S_{\max} + n \le \frac{9}{16} S_{\max}$$

where the last inequality follows from the assumption $S_{\max} = 16n$. By Proposition 10, $s_{i,t} \le s_{i,0} + 2$ for each $t \in [L]$. Therefore, for each $t \in [L]$,

$$\sum_{i=1}^{n} s_{i,t} \le \sum_{i=1}^{n} (s_{i,0} + 2) \le \frac{9}{16} S_{\max} + 2n = \frac{11}{16} S_{\max} \le S_{\max}.$$

For the lower bound on the number of blocks, we use the property from Proposition 10 that $s_{i,t} \ge s_{i,0}$ for each $t \in [L]$: $\sum_{i=1}^{n} s_{i,t} \ge \sum_{i=1}^{n} s_{i,0} \ge \sum_{i=1}^{n} (q_{i,0}/L) \ge \frac{1}{L} \cdot \frac{m}{2} = \frac{1}{4} S_{\max}$. ◀

Given the notion of blocks and its properties, we split the set (of nodes) $[n]$ at round t into *groups* $B_{0,t}, B_{1,t}, \ldots, B_{\log S_{\max},t}$ according to the numbers of blocks such that

$$B_{l,t} = \{i \in [1,n] \mid 2^{l-1} \leq \lceil (q_{i,t}+t)/L \rceil < 2^l\} \tag{5}$$

for $l > 0$ and $B_{0,t}$ is the set of nodes $i \in [n]$ such that $q_{i,t} + t < L$, i.e., the nodes with just one block. Thus, $i \in B_{l,t}$ iff the number of blocks of node i in round t satisfies the inequality $2^{l-1} \leq s_{i,t} < 2^l$. For the sake of brevity, we say that a node i belongs to the group B_l in rounds t iff i is in the group $B_{l,t}$. Proposition 10 implies that each node $i \in B_{j,_}$ can only move to $B_{j+1,_}$ and then to $B_{j+2,_}$. Using (5) and Proposition 11, we can bound the number S_t of blocks at round t from above:

$$S_{\max}/4 \leq S_t = \sum_{i=1}^{n} s_{i,t} \leq \sum_{j=0}^{\log S_{\max}} |B_{j,t}| 2^j \tag{6}$$

and below

$$S_{\max} \geq S_t = \sum_{i=1}^{n} s_{i,t} \geq \sum_{j=0}^{\log S_{\max}} |B_{j,t}| 2^{j-1} \tag{7}$$

Now, given the column/round t, we would like to estimate the probability that the inequality $q_{i,t} \geq M_{i,t}$ holds for exactly one $i \in [n]$ (i.e., t is successful). For a node $i \in B_{l,t}$, we have $2^{l-1} \leq \lfloor (q_{i,t}+t)/L \rfloor < 2^l$. Given the possible values of $M_{i,t}$ (see equality (4)), the highest value of $M_{i,t}$ that is no larger than $q_{i,t}$ is $M_{i,t} = L \cdot 2^l - t$. Therefore, we have

$$\text{Prob}[q_{i,t} \geq M_{i,t}] = \text{Prob}[M_{i,t} \leq L \cdot 2^l - t] = \sum_{k=1}^{l-1} \frac{2^k}{cS_{\max}} \in \left[\frac{2^{l-1}}{cS_{\max}}, \frac{2^l}{cS_{\max}}\right]. \tag{8}$$

Let $p_t = \sum_{i=1}^{n} \text{Prob}[q_{i,t} \geq M_{i,t}]$, i.e., p_t is the expected number of nodes i such that $q_{i,t} \geq M_{i,t}$. Using (8) and (6), we obtain the following bounds:

$$\begin{aligned} p_t &= \sum_{i=1}^{n} \text{Prob}[q_{i,t} \geq M_{i,t}] \geq \sum_{j=1}^{\log S_{\max}} |B_{j,t}| \frac{2^{j-1}}{c \cdot S_{\max}} \geq \frac{1}{2c \cdot S_{\max}} \sum_{j=0}^{\log S_{\max}} |B_{j,t}| 2^j - |B_{0,t}| \\ &\geq \frac{1}{2c \cdot S_{\max}} \cdot \left(\frac{S_{\max}}{4} - |B_{0,t}|\right) \geq \frac{1}{16c} \end{aligned}$$

where the last inequality follows from the fact that $S_{\max} = 16n$ and $|B_{0,t}| \leq n$. On the other hand

$$\begin{aligned} p_t &= \sum_{i=1}^{n} \text{Prob}[q_{i,t} \geq M_{i,t}] \leq \sum_{j=1}^{\log S_{\max}} |B_{j,t}| \frac{2^j}{c \cdot S_{\max}} \leq \frac{2}{c \cdot S_{\max}} \cdot \sum_{j=1}^{\log S_{\max}} |B_{j,t}| 2^{j-1} \\ &\leq \frac{2}{c \cdot S_{\max}} S_{\max} = \frac{2}{c}. \end{aligned}$$

Recall that $c = 4$. It is well known (see e.g. [20]) that given independent events $E_1, \ldots, E_n$ such that $x \leq \sum_{i=1}^{n} \text{Prob}[E_i] \leq \frac{1}{2}$, the probability that exactly one of the events $E_1, \ldots, E_n$ is satisfied is at least $x(1/4)^x$. Thus, in our case, the probability that exactly one of the events $q_{i,t} \geq M_{i,t}$ occurs is at least $\frac{1}{16c}(1/4^{1/(16c)}) \geq d$ for $d = \frac{1}{4^4}$. Let $X = |\{t \in [L] \mid t \text{ is successful}\}|$ denote the number of successful "rounds" in $[L]$ for fixed values of $q_{i,0}$ and $A_{i,t}$ for $i \in [n]$ and $t \in [L]$. The expected value of X is $EX = dL$. For the sake of derandomization, we would like to show that $X \geq \frac{1}{2}EX = \frac{1}{2}dL$ with probability $\geq 1 - 1/2^{-\Omega(L)}$.

Observe that $q_{i,t} \geq M_{i,t}$ for $M_{i,t} = L \cdot 2^k - t$ iff $i \in \bigcup_{j=k}^{\log S_{\max}} B_{j,t}$. Let us define a *scenario* as a fixed sequence of partitions of $[n]$ into groups $B_{0,t}, \ldots, B_{n,t}$ for all $t \in [L]$ which might appear in our setting. There are at most $(1 + \log S_{max})^n$ initial partitions into groups $B_{0,0}, \ldots, B_{n,0}$, since each $j \in [n]$ is in exactly one block from $B_{i,0}, \ldots, B_{i,\log S_{\max}}$. Moreover, each $i \in [n]$ can change its group at most twice (increase by one – see Prop. 10). One can encode such changes of (a node) $i \in [n]$ by (at most) two numbers in $[L]$ determining indices in $[L]$ of (possible) increases of the index of the block at some rounds $t_1, t_2 \in [L]$. Thus, there are at most $(1 + \log S_{max})^n \cdot L^{2n}$ possible scenarios. For a fixed scenarios, the success probabilities $\text{Prob}[|\{i \,|\, q_{i,t} \geq M_{i,t}\}| = 1]$ for various rounds $t \in [L]$ are independent. Therefore, we can use Chernoff Bound to estimate the probability that there are less than $dL/2$ successful rounds:

$$\text{Prob}[X \leq dL/2] \leq \text{Prob}[X \leq (1 - 1/2)EX] \leq e^{-dL/8}.$$

Now, we would like to estimate the proabability (for a matrix M chosen randomly as described above) that there are at least $dL/2$ successful rounds $t \in [L]$ for any scenario. By the union bound, the probability that there exists a scenario with less than $dL/2$ successful rounds is at most

$$e^{-dL/8} \cdot (1 + \log S_{max})^n \cdot L^{2n} < e^{-dL/8} \cdot (17n)^n \cdot L^{2n} = e^{-dL/8 + n\ln(17n) + 2n\ln L} < 1$$

for each $L \geq L_0$ such that $L_0 = O(n \log n)$. Thus, there exists a matrix M which guarantees $dL/2$ successful rounds, provided that $\sum_{i=1}^{n} q_{i,0} \in [m/2, m]$ as assumed in Case 2. If ρ and L are such that $\rho L + b \leq dL/2$, then $\sum_{i=1}^{n} \sum_{t=1}^{L} A_{i,t} \leq \rho L + b \leq dL/2$ is smaller than

$$|\{t \in [L] \,|\, q_{i,t} \geq M_{i,t} \text{ for exactlny one } i \in [n]\}| \geq dL/2.$$

Thus, given the constants $d = 1/4^4$, $\rho < d/2$, any $L \geq L_1$ for $L_1 = O(n \log n + \frac{b}{d/2-\rho}) = O(n \log n + b)$ guarantees $\sum_{i=1}^{n} q_{i,L} \leq \sum_{i=1}^{n} q_{i,0}$, which finishes the proof of Lemma 9.

5 Conclusions

We investigated what stability guarantees we could get in a system with contention if protocols have very limited (or no) space to store information inherited from history of computation and communication. A natural research direction would be to prove tight bound on injection rates and optimize other measures, such as packet latency. Schedulers could also be studied in the context of other related models, such as SINR or dependency-graph models. We introduced a novel class of selectors with thresholds, unlike the previously used binary selectors/codes – studying their constructiveness in polynomial time and further applicability is a prospective open direction.

References

1. Norman M. Abramson. Development of the ALOHANET. *IEEE Transactions on Information Theory*, 31(2):119–123, 1985.
2. L. Anantharamu, Bogdan S. Chlebus, and Mariusz A. Rokicki. Adversarial multiple access channel with individual injection rates. In *Proceedings of the 13th International Conference on Principles of Distributed Systems (OPODIS)*, LNCS 5923, pages 174–188. Springer-Verlag, 2009.
3. Lakshmi Anantharamu, Bogdan S. Chlebus, Dariusz R. Kowalski, and Mariusz A. Rokicki. Deterministic broadcast on multiple access channels. In *Proceedings of the 29th IEEE International Conference on Computer Communications (INFOCOM)*, pages 1–5, 2010.

4 Matthew Andrews, Baruch Awerbuch, Antonio Fernández, Frank Thomson Leighton, Zhiyong Liu, and Jon M. Kleinberg. Universal-stability results and performance bounds for greedy contention-resolution protocols. *Journal of the ACM*, 48(1):39–69, 2001.

5 Michael A. Bender, Martin Farach-Colton, Simai He, Bradley C. Kuszmaul, and Charles E. Leiserson. Adversarial contention resolution for simple channels. In *Proceedings of the 17th Annual ACM Symposium on Parallel Algorithms (SPAA)*, pages 325–332, 2005.

6 Marcin Bieńkowski, Marek Klonowski, Mirosław Korzeniowski, and Dariusz R. Kowalski. Dynamic Sharing of a Multiple Access Channel. In *Proceedings of the 27th International Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 83–94, 2010.

7 Allan Borodin, Jon M. Kleinberg, Prabhakar Raghavan, Madhu Sudan, and David P. Williamson. Adversarial queuing theory. *Journal of the ACM*, 48(1):13–38, 2001.

8 Bogdan S. Chlebus, Dariusz R. Kowalski, Andrzej Pelc, and Mariusz A. Rokicki. Efficient Distributed Communication in Ad-Hoc Radio Networks. In *Proceedings of the 38th International Colloquium on Automata, Languages and Programming (ICALP), Part II*, volume 6756 of *Lecture Notes in Computer Science*, pages 613–624. Springer, 2011.

9 Bogdan S. Chlebus, Dariusz R. Kowalski, and Mariusz A. Rokicki. Maximum throughput of multiple access channels in adversarial environments. *Distributed Computing*, 22(2):93–116, 2009.

10 Bogdan S. Chlebus, Dariusz R. Kowalski, and Mariusz A. Rokicki. Adversarial Queuing on the Multiple Access Channel. *ACM Transactions on Algorithms*, 8(1):5:1–5:31, 2012.

11 Jurek Czyżowicz, Leszek Gąsieniec, Dariusz R. Kowalski, and Andrzej Pelc. Consensus and Mutual Exclusion in a Multiple Access Channel. In *Proceedings of the 23rd International Symposium on Distributed Computing (DISC)*, LNCS 5805, pages 512–526. Springer-Verlag, 2009.

12 Sergey Foss. Lectures on Stochastic Stability. Abo Akademi University, June 2008. URL: http://web.abo.fi/fak/mnf/mate/tammerfors08/Foss_Lecture2.pdf.

13 Robert G. Gallager. A perspective on multiaccess channels. *IEEE Transactions on Information Theory*, 31(2):124–142, 1985.

14 Pawel Garncarek, Tomasz Jurdzinski, and Dariusz R. Kowalski. Local Queuing Under Contention. In *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018*, pages 28:1–28:18, 2018. doi:10.4230/LIPIcs.DISC.2018.28.

15 Leslie Ann Goldberg, Mark Jerrum, Sampath Kannan, and Mike Paterson. A bound on the capacity of backoff and acknowledgment-based protocols. *SIAM Journal on Computing*, 33(2):313–331, 2004.

16 Leslie Ann Goldberg, Philip D. MacKenzie, Mike Paterson, and Aravind Srinivasan. Contention resolution with constant expected delay. *Journal of the ACM*, 47(6):1048–1096, 2000.

17 Albert G. Greenberg and Shmuel Winograd. A Lower Bound on the Time Needed in the Worst Case to Resolve Conflicts Deterministically in Multiple Access Channels. *Journal of the ACM*, 32(3):589–596, 1985.

18 Johan Håstad, Frank Thomson Leighton, and Brian Rogoff. Analysis of Backoff Protocols for Multiple Access Channels. *SIAM J. Comput.*, 25(4):740–774, 1996. doi:10.1137/S0097539792233828.

19 Tomasz Jurdziński, Miroslaw Kutyłowski, and Jan Zatopiański. Efficient algorithms for leader election in radio networks. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing (PODC)*, pages 51–57, 2002.

20 Tomasz Jurdzinski and Grzegorz Stachowiak. Probabilistic Algorithms for the Wake-Up Problem in Single-Hop Radio Networks. *Theory Comput. Syst.*, 38(3):347–367, 2005. doi:10.1007/s00224-005-1144-3.

21 János Komlós and Albert G. Greenberg. An asymptotically fast nonadaptive algorithm for conflict resolution in multiple-access channels. *IEEE Transactions on Information Theory*, 31(2):302–306, 1985.

22 Dariusz R. Kowalski. On selection problem in radio networks. In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 158–166, 2005.

23 Eyal Kushilevitz and Yishay Mansour. An $\Omega(D \log(N/D))$ Lower Bound for Broadcast in Radio Networks. *SIAM Journal on Computing*, 27(3):702–712, 1998.

24 Robert M. Metcalfe and David R. Boggs. Ethernet: Distributed Packet Switching for Local Computer Networks. *Communications of the ACM*, 19(7):395–404, 1976.

25 Prabhakar Raghavan and Eli Upfal. Stochastic Contention Resolution With Short Delays. *SIAM Journal on Computing*, 28(2):709–719, 1998.

26 Dan E. Willard. Log-Logarithmic Selection Resolution Protocols in a Multiple Access Channel. *SIAM Journal on Computing*, 15(2):468–477, 1986.

Improved Network Decompositions Using Small Messages with Applications on MIS, Neighborhood Covers, and Beyond

Mohsen Ghaffari
ETH Zürich, Switzerland
ghaffari@inf.ethz.ch

Julian Portmann
ETH Zürich, Switzerland
pjulian@ethz.ch

Abstract

Network decompositions, as introduced by Awerbuch, Luby, Goldberg, and Plotkin [FOCS'89], are one of the key algorithmic tools in distributed graph algorithms. We present an improved deterministic distributed algorithm for constructing network decompositions of power graphs using small messages, which improves upon the algorithm of Ghaffari and Kuhn [DISC'18]. In addition, we provide a randomized distributed network decomposition algorithm, based on our deterministic algorithm, with failure probability exponentially small in the input size that works with small messages as well. Compared to the previous algorithm of Elkin and Neiman [PODC'16], our algorithm achieves a better success probability at the expense of its round complexity, while giving a network decomposition of the same quality. As a consequence of the randomized algorithm for network decomposition, we get a faster randomized algorithm for computing a Maximal Independent Set, improving on a result of Ghaffari [SODA'19]. Other implications of our improved deterministic network decomposition algorithm are: a faster deterministic distributed algorithms for constructing spanners and approximations of distributed set cover, improving results of Ghaffari, and Kuhn [DISC'18] and Deurer, Kuhn, and Maus [PODC'19]; and faster a deterministic distributed algorithm for constructing neighborhood covers, resolving an open question of Elkin [SODA'04].

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Distributed Graph Algorithms, Network Decomposition, Maximal Independent Set, Neighborhood Cover

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.18

Related Version A full version of the paper is available at https://arxiv.org/abs/1908.03500.

Funding The authors' research is supported by the Swiss National Foundation, under project number 200021_184735.

1 Introduction

We present an improved deterministic distributed algorithm for constructing network decompositions of power graphs using small messages, as well as some improvements for other problems including randomized construction of maximal independent set, and deterministic construction of sparse neighborhood covers, spanners and dominating set approximation.

After introducing our model of computation, we recall the concept of network decompositions in Section 1.1 as well as a brief summary of all known distributed constructions. In Section 1.2 we present our results and in Section 1.3 we outline our methods and explain how they depart from previous approaches.

© Mohsen Ghaffari and Julian Portmann;
licensed under Creative Commons License CC-BY
33rd International Symposium on Distributed Computing (DISC 2019).
Editor: Jukka Suomela; Article No. 18; pp. 18:1–18:16

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Model. Throughout, we work with the CONGEST model of distributed computing [27]: The communication network is abstracted as an n-node graph $G = (V, E)$. We use Δ to denote the maximum degree of G. There is one processor on each node of the network, which initially knows only its $O(\log n)$-bit identifier. Per round of synchronous communication, every node can send one $O(\log n)$-bit message to each neighbor. Note that this is enough to describe constantly many elements of the network, i.e. vertices or edges. A closely related variant is the LOCAL model [22], where we impose no restriction on the size of messages.

1.1 Network Decompositions

Network decompositions were introduced by Awerbuch et al. [8], and since then, they have turned out to be one of the key algorithmic tools in distributed algorithms for graph problems. For a given graph $G = (V, E)$, a (c, d) network decomposition of it is defined as a partition of V into *blocks* $V_1, \ldots, V_c$ such that each connected component of the subgraphs $G[V_i]$ has diameter at most d. The connected components of each block are usually called *clusters*. This notion of network decomposition is sometimes also referred to as *strong diameter* network decomposition, as we consider the diameter with respect to distances in the induced subgraphs. This is as opposed to *weak diameter* network decompositions, where distances are with respect to the base graph. Intuitively, network decompositions allow us to process graph problems in c sequential stages, where in each stage we process one block, a graph that is made of low-diameter components (diameter d). This low-diameter simplifies the task as it opens the road for collecting either the entire topology, in the LOCAL model, or at least some coordination messages, in the CONGEST model. The key point is that the problems in different components of one block can be processed independently, as they have distance at least 1.

In many applications of network decompositions, instead of asking for the clusters to have distance at least 1, we need them to have a larger distance, at least k hops for some parameter $k \geq 2$. This is crucial for applications where the problem is such that the answer in one node can impact nodes beyond its neighbors. Thus, a natural extension of network decomposition is the following: a k-hop separated network decomposition or decomposition of G^k requires that any two nodes u, v from different clusters of the same color are at distance more than k in G. We note that clusters do not have to be connected in G, which means that it is a *weak diameter* decomposition of G.

While the authors of [8] used network decompositions to solve symmetry breaking problems, such as maximal independent set or $(\Delta + 1)$-vertex coloring, various other applications were discovered later. Examples in the LOCAL model include the computation of sparse spanners and linear-size skeletons by Dubhashi et al. [16] or distributed approximation algorithms for the graph coloring and minimum dominating set problems by Barenboim et al. [10, 11]. For the CONGEST model, Ghaffari and Kuhn [21] showed that k-hop separated network decompositions can be used for computing spanners and approximating minimum dominating set.

State of the Art – Deterministic Constructions. There are four known deterministic distributed constructions of network decompositions, successively improving either quantitatively or qualitatively [8, 20, 21, 26]. Awerbuch et al. [8] provided an algorithm for computing $(2^{O(\sqrt{\log n \log \log n})}, 2^{O(\sqrt{\log n \log \log n})})$ network decompositions of an n node graph G in $2^{O(\sqrt{\log n \log \log n})}$ rounds, which works in the CONGEST model. Subsequently, this was improved by Panconesi and Srinivasan [26] showing that all $2^{O(\sqrt{\log n \log \log n})}$ terms could be replaced by $2^{O(\sqrt{\log n})}$. However, their algorithm requires large messages.

For computing network decompositions with higher levels of separation, Ghaffari and Kuhn [21] gave a $k \cdot 2^{O(\sqrt{\log n \log\log n})}$ round CONGEST-model algorithm for computing a $(2^{O(\sqrt{\log n \log\log n})}, 2^{O(\sqrt{\log n \log\log n})})$ network decomposition of G^k, which works with small messages. Note that extending network decomposition algorithms to compute a decomposition of G^k is trivial in the LOCAL model: As nodes can send messages of arbitrary size, communication on G^k can be simulated in k rounds of communication on G. Thus, with a k factor overhead in the round complexity (and a k factor increase in the diameter with respect to distances in G), we can use any LOCAL-model network decomposition algorithm to also compute k-hop separated decompositions.

Recently, Ghaffari [20] showed that a $(2^{O(\sqrt{\log n})}, 2^{O(\sqrt{\log n})})$ network decomposition can also be computed in $2^{O(\sqrt{\log n})}$ rounds in the CONGEST model. However, his construction cannot extend to G^k, which is one of the issues we address in this paper. In contrast to all previous approaches, this algorithm has the useful property, that it can handle large identifiers. This means that the length of identifiers does not influence the parameters of the resulting network decomposition.

State of the Art – Randomized Constructions. For randomized algorithms, there are stronger results: Linial and Saks [23] showed that $(O(\log n), O(\log n))$ network decompositions exist and gave a distributed algorithm, which finds a $(O(\log n), O(\log n))$ network decomposition in $O(\log^2 n)$ rounds, with high probability[1] (w.h.p). The construction of Linial and Saks [23] only guarantees that clusters have weak diameter $O(\log n)$. More recently, Elkin and Neiman [18] provided a randomized distributed algorithm that computes strong diameter $(O(\log n), O(\log n))$ network decomposition in $O(\log^2 n)$, w.h.p, and also works in the CONGEST model. Both of these algorithms can be easily extended to produce a $(O(\log n), O(k \log n))$ decomposition of G^k in $O(k \log^2 n)$ rounds without requiring larger messages.

We remark that the fact that these algorithm succeed with probability $1 - 1/\text{poly}(n)$ prevents them from being directly used in our randomized MIS algorithm. This is because after the shattering, only components of size $N \ll n$ remain, which means that the algorithms only succeed with probability $1 - 1/\text{poly}(N)$ in computing a $(O(\log N), O(\log N))$ network decomposition.

1.2 Our Results

We present a deterministic distributed CONGEST-model algorithm for computing network decompositions of G^k:

▶ **Theorem 1.** *There is a deterministic distributed algorithm that in any N-node network G, which has S-bit identifiers and supports $O(S)$-bit messages for some arbitrary S, computes a $(g(N), g(N))$ network decomposition of G^k in $kg(N) \cdot \log^* S$ rounds, for any k, and $g(N) = 2^{O(\sqrt{\log N})}$.*

We highlight the following three properties, whose combination is new to our algorithm and is crucial for our applications in the next subsections: (A) it is able to compute a network decomposition of G^k in the CONGEST model, (B) it can handle large identifiers, and (C)

[1] As usual, we use the phrase *with high probability* to denote that an event holds with probability at least $1 - n^{-c}$ for any constant c, where c may influence other constants.

its bounds are as good as a simulation of the algorithm of [26] on G^k in the LOCAL model. More precisely, property (B) says that the size of identifiers only affects the round complexity but not the quality of the computed network decomposition.

In particular for our application to MIS, property (B) is crucial. The lack of this ability to handle large identifiers is what made previous algorithms, such as of Ghaffari and Kuhn [21], not applicable. We refer to Section 1.3 for a more in-depth explanation of these issues.

Applications: MIS, Neighborhood Cover, and Beyond

Network decompositions have a wide range of applications and due to previous work, our new algorithms leads to an improvement for a number of problems. While some of these results are immediate, for the application to MIS, we also present a randomized algorithm for network decompositions whose failure probability is exponentially small in the input size.

The MIS Problem. The Maximal Independent Set Problem (MIS) asks for a set S of nodes, such that no two neighboring nodes are in S and moreover, for each node v, either v or at least one of its neighbors is in S. It is one of the most well-studied distributed graph problems. One reason for its importance is that other fundamental graph problems, such as $(\Delta + 1)$-vertex coloring, maximal matching, or 2-approximation of vertex cover reduce to it [22, 24].

Luby [24] as well as Alon, Babai and Itai [2] gave randomized distributed MIS algorithms in the CONGEST model that have round complexity $O(\log n)$. The first significant improvement over this run time was due to Barenboim, Elkin, Pettie, and Schneider [12], who gave a randomized distributed $O(\log^2 \Delta) + 2^{O(\sqrt{\log \log n})}$ round algorithm. This bound was then improved by Ghaffari [19] to $O(\log \Delta) + 2^{O(\sqrt{\log \log n})}$, which remains the state of the art. However, both these improvements do not work in the CONGEST model, as they require messages of up to $\text{poly}(\Delta, \log n)$ bits to gather certain local topologies. The only improvement upon the algorithms of [2, 24] in the CONGEST model is due to Ghaffari [20], who gave a randomized distributed algorithm that runs in $\min\{\log \Delta \cdot 2^{O(\sqrt{\log \log n})}, O(\log \Delta \cdot \log \log n) + 2^{O(\sqrt{\log \log n \cdot \log \log \log n})}\}$ rounds.

We improve this result for all values of Δ and obtain the following:

▶ **Theorem 2.** *There is a randomized distributed algorithm, with $O(\log n)$-bit messages, that computes an MIS in $O\big(\log \Delta \cdot \sqrt{\log \log n}\big) + 2^{O(\sqrt{\log \log n})}$ rounds, w.h.p.*

Apart from our improved network decomposition, this result contains a randomized algorithm, that transforms a network decomposition of G^k into a decomposition of G with improved parameters. This transformation works in the CONGEST model and succeeds with probability exponential in the input size, which is crucial for its application in solving MIS. For a more detailed overview, see Section 1.3.

Neighborhood Covers and MST. Neighborhood covers, as introduced by Awerbuch and Peleg [4] are another form of locality-preserving graph representations and closely related to network decompositions. A s-sparse k-neighborhood cover of diameter d is defined as a collection of clusters $C \subseteq V$ such that (A) for each cluster C, we have a rooted spanning tree of $G[C]$ with diameter at most d, (B) each k-neighborhood of G is completely contained in some cluster, and (C) each node of G is in at most s clusters. Like network decompositions, this form of graph representation has many applications in distributed computing, such as in routing [9], shortest paths [1], job scheduling and load balancing [7], or broadcast and multicast [6].

Awerbuch and Peleg [4] also gave distributed constructions for sparse neighborhood covers using messages of unbounded size, however they do not extend to the CONGEST model. More recently, Ghaffari and Kuhn [21] gave the first CONGEST model algorithm for computing sparse neighborhood covers. They showed that a (c, d) network decomposition of G^{2k} can be transformed into a c-sparse k-neighborhood cover of diameter $O(k \cdot d)$ in $O(c(d + k))$ rounds. Together with their $k \cdot 2^{O(\sqrt{\log n \log \log n})}$ round algorithm for computing a $(2^{O(\sqrt{\log n \log \log n})}, 2^{O(\sqrt{\log n \log \log n})})$ network decomposition of G^k, this yields a $k \cdot 2^{O(\sqrt{\log n \log \log n})}$ round algorithm for computing $2^{O(\sqrt{\log n \log \log n})}$-sparse k-neighborhood covers of diameter $k \cdot 2^{O(\sqrt{\log n \log \log n})}$.

Using our network decomposition, we improve upon the result of [21] and show that all $2^{O(\sqrt{\log n \log \log n})}$ terms can be replaced by $2^{O(\sqrt{\log n})}$. The proof is deferred to the full version of the paper.

▶ **Corollary 3.** *There is a deterministic distributed algorithm, that for every $k \geq 1$, computes a $2^{O(\sqrt{\log n})}$-sparse k-neighborhood cover of diameter $k \cdot 2^{O(\sqrt{\log n})}$ of and n-node graph G in $k \cdot 2^{O(\sqrt{\log n})}$ rounds of* CONGEST.

We also resolve an open question by Elkin [17], who devised a randomized CONGEST model algorithm for minimum spanning tree, that runs in $\tilde{O}(\mu(G, \omega) + \sqrt{n})$ rounds, where $\mu(G, \omega)$ is the *MST-radius* of G. The MST-radius $\mu(G, \omega)$ is defined as the smallest value t, such that every edge not belonging to the MST of G is the heaviest edge in some cycle of length at most t. However, the only part involving randomness is the construction of neighborhood covers. The author remarks that the only obstacle towards a deterministic algorithm is that there are no known constructions of sparse neighborhood covers in the CONGEST model. Using Corollary 3, we get a deterministic distributed CONGEST-model algorithm for computing MST in $2^{O(\sqrt{\log n})} \cdot (\mu(G, \omega) + \sqrt{n})$ rounds. For a discussion on how to use Sparse Neighborhood Covers in computing MST, we refer to [17] or the full version of this paper.

Other Problems: Spanners and Dominating Set Approximation. Due to previous applications of k-hop separated network decompositions by Ghaffari and Kuhn [21] as well as Deurer, Kuhn, and, Maus [15] we obtain the following deterministic CONGEST model algorithms: A $2^{O(\sqrt{\log n})}$ round algorithm for computing a $(2k - 1)$-stretch spanner with size $O(kn^{1+1/k} \log n)$, and a $O(\log \Delta)$-approximation algorithm for minimum dominating set in $2^{O(\sqrt{\log n})}$ rounds. For a discussion, see the full version of this paper.

1.3 Method Overview and Comparison with Prior Approaches

We first discuss our method for deterministic network decomposition, and then discuss our contribution to the MIS problem.

Network Decomposition. The general outline is shared by all known deterministic algorithms for network decomposition [8, 20, 21, 26]. This method is often referred to as recursive clustering: In every step, a number of clusters is merged to form new clusters while some other clusters are added to the output and discarded from the algorithm. However, there are several challenges in applying this approach in the CONGEST model, and even more so when aiming to compute a decomposition of G^k.

Let us address these challenges in two themes, (A) communication within clusters, and (B) communication between clusters: Our approach entails the fact that clusters can become disconnected in the base graph G, even if they are connected in G^k. While this means that

we have more freedom in how we merge clusters, it also requires us to take extra care to allow for intra-cluster communication. As clusters can now "overlap", a single edge of G could be used by many clusters for communication. We will have a two stage process to "introduce" clusters to their neighboring clusters (in G^k), which will enable us to bound the "overlap" between clusters. To be more precise we will argue that any edge is used by at most $2^{O(\sqrt{\log n})}$ many clusters for communication. This allows us to have simultaneous communication in all clusters with just a $2^{O(\sqrt{\log n})}$ factor overhead.

For challenge (B), we would like to simulate communication in G_C on G, where G_C is the virtual graph obtained by contracting each cluster into a node and connecting two clusters if they contain nodes that are adjacent in G^k. However, this simulation is not directly possible in the CONGEST model, as nodes in G_C can have degree much larger than Δ, leading to congestion for communication within clusters. The solution to this problem will also be the introduction process mentioned above. Roughly speaking, we will ignore some edges from G_C as well as some vertices of high degree. This allows for an efficient simulation of communication in G_C on the base graph G.

Maximal Independent Set. For solving MIS, we follow the outline of the *shattering technique*, first introduced into distributed computing by Barenboim et al. [12], and also used for the MIS problem by [19, 20]. There are two parts: In the pre-shattering phase, we solve the problem for a large portion of the graph, leaving only a number of "small" connected components. Then, in the post-shattering phase, we solve the problem on the remaining parts.

In the pre-shattering phase, we use the $O(\log \Delta)$-round algorithm of Ghaffari [19], which works with just single-bit messages. Afterwards, we are left with "small" components. For now, assume that they have size[2] $O(\log n)$. By computing a network decomposition of each component, we can further simplify the problem: We go through the color classes, one by one, each time computing an MIS of the new color, that does not conflict with the MIS of the previous colors. We solve the problem by running $O(\log n)$ independent copies of Ghaffari's $O(\log \Delta)$-round randomized MIS algorithm algorithm [19], all in parallel. This parallel execution is possible in the CONGEST model because the algorithm from [19] only uses single-bit messages. With high probability (i.e. at least $1 - 1/\text{poly}(n)$), at least one of these independent runs succeeds in computing an MIS. Using the fact that we are solving the problem in a graph of low diameter, we can efficiently coordinate all nodes to find a successful run.

The main challenge is obtaining a suitable network decomposition: We could use the network decomposition algorithm from Theorem 1 and get an MIS algorithm with round complexity $\log \Delta \cdot 2^{O(\sqrt{\log \log n})}$. This only matches the previous work of Ghaffari [20]. Also, randomized algorithms for network decomposition are hard to apply, as we are computing decompositions of graphs that only contain $N = O(\log n)$ nodes. This means that the success probability of randomized algorithms for network decomposition, such as [18, 23], will only be $1 - 1/\text{poly}(N) \ll 1 - 1/\text{poly}(n)$.

We get around these issues in two steps: First, we compute a k-hop separated network decomposition of each component. Then, we use this network decomposition to boost the success probability of a randomized network decomposition algorithm, inspired by [14, 18, 25]. While Ghaffari [20] used a similar idea to also get an $O(\log \Delta \cdot \log \log n) + 2^{O(\sqrt{\log \log n \cdot \log \log \log n})}$

[2] They do not contain $O(\log n)$ nodes, but rather up to $O(\Delta^4 \log n)$ many vertices. However, we will see that we can efficiently cluster them into $O(\log n)$ clusters of diameter only $O(\log \log n)$.

round algorithm for MIS, we improve upon it in two ways: First, our network decomposition of G^k has better bounds, and second, we use a randomized process for computing a refined network decomposition. This randomized process allows us to further reduce the number of colors needed, from $O(\log\log n)$ to $O(\sqrt{\log\log n})$.

1.4 Mathematical Notation

For a graph $G=(V,E)$ and two nodes $u,v\in V$, we define $d_G(u,v)$ to be the hop distance between u and v. For a node $v\in V$ and a set $U\subset V$, $dist_G(v,U)$ is the minimum distance between v and any $u\in U$. For an integer $k\geq 1$ we define the kth *power* $G^k=(V,E')$ of G to be the graph with an edge $\{u,v\}\in E'$ whenever $d_G(u,v)\leq k$. Given a node $v\in V$, we define $N_{G,k}:=\{u\in V: d_G(u,v)\leq k\}$ to be the *k-hop neighborhood* of v.

For two integers $\alpha\geq 1$ and $\beta\geq 0$ and a node set $B\subseteq V$, we call $B^*\subseteq B$ a (α,β)-ruling set of G w.r.t. B if (A) for any two nodes $u,v\in B^*$ we have $d_G(u,v)\geq\alpha$, and (B) $\forall u\in B\setminus B^*$, there is a node $v\in B^*$ such that $d_G(u,v)\leq\beta$. If $B=V$, B^* is simply called an (α,β) ruling set of G.

2 Network Decomposition

In this section we describe our algorithm for computing network decompositions of power graphs in the CONGEST model, as outlined in Theorem 1. It matches the bounds of the algorithm by Panconesi and Srinivasan [26], but improves upon it in three aspects that are crucial to our applications: our algorithm works in the CONGEST model, can tolerate large identifiers and is able to produce a network decomposition of G^k. While the first two properties were already achieved by Ghaffari [20], the third property is new to our approach.

Note that it is trivial to achieve such a decomposition using messages of unbounded size, by just simulating communication in G^k on G (with a k factor overhead). In the CONGEST model this idea is not directly applicable and presents two challenges: (A) how do we deal with clusters being disconnected in the base graph and (B) how do we get around simulating all communication in G^k on G? For the first issue we will bound the number of clusters that are overlapping. For the second issue we will see that not all communication is necessary.

Before proceeding to the algorithm, we restate Theorem 1 in slightly more detail:

▶ **Theorem 4.** *There is a deterministic distributed algorithm that in any N-node network G, which has S-bit identifiers and supports $O(S)$-bit messages for some arbitrary S, computes a $(g(N),g(N))$ network decomposition of G^k in $kg(N)\cdot\log^* S$ rounds, for any k and $g(N)=2^{O(\sqrt{\log N})}$. Additionally we can simulate one round of communication within clusters of G^k in $k2^{O(\sqrt{\log N})}$ rounds of communication on G.*

▶ Remark 5. If we initially have N clusters, each with an S-bit center identifier and with radius at most r, the algorithm of Theorem 4 computes a $(g(N),rg(N))$ network decomposition of G^k in $krg(N)\cdot\log^* S$ rounds.

Proof of Theorem 4. We first note that the recursive nature of the algorithm makes it directly applicable to use with an initial clustering, as described in Remark 5.

Overall Structure. The algorithm consists of phases $i=1,\ldots,\sqrt{\log N}$, each of which runs in $k\cdot 2^{O(\sqrt{\log N})}\cdot\log^* S$ rounds. During each phase, the (remaining) vertices are partitioned into vertex-disjoint clusters. Each cluster has one center node (which will be the identifier of the cluster), as well as a tree rooted at the center that spans all vertices of this cluster

(and potentially also contains vertices of other clusters). However we have that any two nodes of the cluster are connected by a path of length at most k in this tree. Note that both edges and vertices of G can be included in multiple trees.

Initially, every node is its own cluster. During each phase some clusters join each other to form some new clusters, while the other clusters are colored and removed from the algorithm. Let $d = 2^{O(\sqrt{\log N})}$. We maintain the following invariants during the ith phase:

(A) We have at most n/d^i clusters.

(B) The radius of each cluster is at most $h_i = (O(1))^i \leq 2^{O(\sqrt{\log N})}$ in G^k, which means its radius is at most $k \cdot 2^{O(\sqrt{\log N})}$ in G.

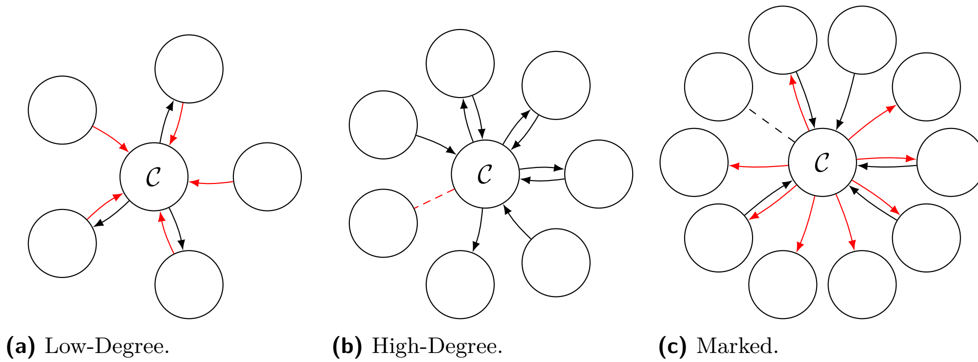
(C) Each edge e is part of at most $i \cdot 13d^3$ spanning trees, which is always at most $2^{O(\sqrt{\log N})}$.

Note that for $i = 0$, these invariants are trivially fulfilled. The first invariant ensures that after $\sqrt{\log N}$ phases, there is at most one cluster left, which we can then color with one color and finish the algorithm. The second invariant means that cluster radii remain small enough. Invariants (B) and (C) together imply that we can perform $2^{O(\sqrt{\log N})} \cdot \log^* S$ iterations of broadcast and convergecast in each cluster within each phase. This is because we can simulate a round of communication along an edge in the spanning tree of G^k within $k \cdot 2^{O(\sqrt{\log N})}$ rounds of communication on G.

Informal Outline of each phase. We start out with a set of (old) clusters and will merge some of them into new clusters, while we color the remaining ones and add them to the resulting decomposition. In a first step, each cluster $\mathcal{C}$ will try to learn its neighboring clusters. If $\mathcal{C}$ has more than $4d^2$ neighbors, we call $\mathcal{C}$ *marked*. If we now consider the graph $\mathcal{G}$ induced by all non-marked clusters, it has maximum degree $\Delta_{\mathcal{G}} \leq 4d^2$. This allows us to simulate communication within $\mathcal{G}$ in the underlying network, with about a $4d^2$ overhead in the round complexity (ignoring cluster diameters). We will use this fact to find a well-separated set $\mathcal{C}^*$ in $\mathcal{G}$. All clusters from $\mathcal{C}^*$, together with the marked clusters will now form the centers of new clusters. Then all old clusters that have a neighboring center join this center to form a new cluster. Intuitively, as all new cluster centers have high degree, there cannot be a lot of them. What we are now left with is a set of low-degree clusters that are not part of any newly formed cluster. We can now just color these remaining clusters, using standard coloring techniques, and add them to the final output. As the degrees are low, the number of required colors is also low.

Building a small in-degree virtual graph H. Call two clusters $\mathcal{C}$ and $\mathcal{C}'$ neighboring if they contain vertices $v \in \mathcal{C}$ and $v' \in \mathcal{C}'$ such that v and v' are at distance at most k in G. This means that v and v' are neighbors in G^k. Similarly, a node v and a cluster $\mathcal{C}$ are called neighboring if there is some $u \in \mathcal{C}$ such that u is at distance at most k from v.

Now we want every cluster to learn about up to $2d$ many neighboring clusters. More precisely, a cluster that has less than $2d$ neighbors should learn about all of its neighbors. If it has more than $2d$ neighbors, it learns about some $2d$ of them. We can do so in $O(k \cdot d)$ rounds: Every node starts a broadcast, sending the identifier of its current cluster to all neighbors. Then, over $(2d+1) \cdot (k-1)$ rounds, every node v forwards up to $2d+1$ different such messages about clusters of distance up to $(k-1)$ from v. This way, if node has at most $2d$ neighboring clusters it learns about all of them and if there are more, it learns about at least $2d$ many, of which it picks some $2d$ many arbitrarily. Within clusters, the nodes convergecast at most $2d$ identifiers to the center node. This is possible in $O((d + k \cdot h_i) \cdot 2^{O(\sqrt{\log N})}) = k \cdot 2^{O(\sqrt{\log N})}$ rounds, as invariant (C) states that at most $2^{O(\sqrt{\log N})}$ clusters overlap. Thus, a $2^{O(\sqrt{\log n})}$ round overhead is enough to allow all clusters to perform a convergecast at the same time.



(a) Low-Degree. **(b)** High-Degree. **(c)** Marked.

Figure 1 Different states of a cluster $\mathcal{C}$ in the virtual graph H, where clusters are vertices and an edge $\mathcal{C} \to \mathcal{C}'$ means that the center of $\mathcal{C}'$ received the identifier of $\mathcal{C}$. Dashed lines indicate that clusters are neighboring, but neither center received the ID of the other center.

This process creates a directed virtual graph H among the clusters, where an edge $\mathcal{C} \to \mathcal{C}'$ indicates that the center of $\mathcal{C}'$ received the identifier of $\mathcal{C}$. We call a cluster *high-degree* if it has at least $2d$ neighboring clusters, and *low-degree* otherwise, see Figure 1. Notice that a low-degree clusters has all neighboring clusters as incoming edges in H. Also, a high-degree cluster has at least $2d$ incoming edges.

Making H undirected with small degrees. One problem is that H is a directed graph with possibly large out-degrees, while we would like to have an undirected graph with small degrees. Additionally we would like to keep the fact that all low degree clusters are adjacent to all their neighboring clusters in this virtual graph. For that, we first *mark* clusters of extremely high out-degree as follows: we reverse the communication direction of the previous paragraph, but instead of sending just one message per round along each edge, we send up to $4d^2$ messages. This increases the number of rounds by at most a $4d^2$ factor. This way, if a message from some cluster $\mathcal{C}$ was sent along an edge in the previous phase, we send up to $4d^2$ messages in the opposite direction, all from clusters that received the identifier of $\mathcal{C}$. If more clusters received the identifier of $\mathcal{C}$, we just inform $\mathcal{C}$ that it will be marked. This can be done within $O(k \cdot d^3) = k \cdot 2^{O(\sqrt{\log N})}$ rounds, as every round from the previous paragraph now takes $4d^2$ as long. Also, at most $(2d+1) \cdot 4d^2 \leq 12d^3$ many messages are sent along each edge. As in the previous paragraph, we can now convergecast the identifiers of at most $4d^2$ outgoing neighbors in $O((d^2 + k \cdot h_i) \cdot 2^{O(\sqrt{\log N})}) = k \cdot 2^{O(\sqrt{\log N})}$ rounds to the cluster centers, marking them the same way as before.

Now, we temporarily remove marked clusters from H; we later discuss how to deal with them. Note that there are at most $\frac{n}{2d^{i+1}}$ many marked clusters. This is because each cluster has in-degree at most $2d$, which means at most a $1/(2d)$ fraction of clusters can have out-degree exceeding $4d^2$. This is at most $\frac{n}{d^i} \cdot \frac{1}{2d}$ many clusters, by invariant (A). We now have an undirected virtual graph on the clusters, which has degree at most $4d^2$.

Computing a Maximal 2-Independent Set in H. H has now degree at most $4d^2$, but we need an additional fact to ensure that we can simulate the communication along H in G: Every edge is part of at most $12d^3 = 2^{O(\sqrt{\log N})}$ edges of H. This is because we can think of every message in the previous phase as trying to establish an edge between two clusters $\mathcal{C}$, $\mathcal{C}'$ in H. Such an edge is only established if a message from $\mathcal{C}$ actually reaches $\mathcal{C}'$. As every edge in G only forwarded $12d^3$ many such messages, it can only be part of as many edges in H.

This means that we can now simulate one round of CONGEST model on H in $O(k \cdot d^3 + (d^2 + k \cdot h_i) \cdot 2^{O(\sqrt{\log n})}) = k \cdot d^3 \cdot 2^{O(\sqrt{\log N})}$ rounds of the base graph. This is because every edge is additionally only part of at most $2^{O(\sqrt{\log N})}$ clusters, by our invariant (C). Using that, we first compute a coloring of H^2, hence ensuring that any two clusters that are within 2 hops in H have different colors. That can be done in $O(d^8 \log^* S \cdot (k \cdot d^3 \cdot 2^{O(\sqrt{\log n})}) = O(d^{11} \log^* S \cdot 2^{O(\sqrt{\log N})})$ rounds, using Linial's algorithm [22], which runs in $O(\Delta_H^2 \log^* S)$ rounds, as H^2 has maximum degree $\Delta_H = O(d^4)$. Then, we compute a maximal 2-independent set C^* of high-degree clusters (this is the definition of high degree mentioned above, which is with respect to the neighborhood of clusters in G^k). Here, 2-independent set means that no two clusters in C^* should share a common neighboring cluster in H. We can do so by going through all colors one by one, adding clusters to C^* that do not already have a cluster from C^* within distance two in H. Any $\mathcal{C}$ that has a $\mathcal{C}' \in C^*$ within its 2-cluster-hops joins the new cluster being formed at the center of $\mathcal{C}'$. As all high-degree clusters not in C^* must have a neighbor in C^* within 2-cluster hops, all high-degree clusters are part of a newly formed cluster.

Forming new clusters. Each high-degree cluster $\mathcal{C}' \in C^*$ has two cases: (I) either none of the neighboring clusters of $\mathcal{C}'$ were marked, in which case all of them will join the new cluster being formed by $\mathcal{C}'$. This means that the new cluster contains at least $2d$ many old clusters. Thus, there are at most $\frac{n}{d^i} \cdot \frac{1}{2d}$ many such new clusters. (II) at least one of the neighboring clusters of $\mathcal{C}'$ was marked. In this case, after $\mathcal{C}'$ accepts the clusters that want to join with it, $\mathcal{C}'$ picks one of its marked neighbors and joins a new cluster centered at that marked cluster. To make clusters learn about neighbors, use k rounds of flooding, initiated at all nodes of marked clusters. This way, we have to send the identifier of at most one marked cluster along each edge, to ensure that all clusters know if they have a marked neighbor. Since there are at most $\frac{n}{2d^{i+1}}$ many marked clusters, the number of the new clusters of this kind is also at most $\frac{n}{2d^{i+1}}$.

Proving the inductive invariants. By the previous paragraph, we have at most $\frac{n}{d^{i+1}}$ many new clusters, proving invariant (A). Regarding invariant (B), first notice that each new cluster that we form is made of some of the previous clusters, all of which were within $O(1)$ cluster hops (w.r.t. distances in G^k) of the center of the merge (either in C^* or a marked cluster). Hence, the maximum cluster radius grows by at most a factor of $O(1)$, which shows that each cluster radius in phase i is at most $(O(1))^{i+1}$ in G^k.

For invariant (C), we have already argued that due to merges between non marked clusters, every edge is used by at most $12d^3$ many additional clusters, as these merges only happen along edges of H. For the merging centered at a marked cluster $\mathcal{C}$, we have that if an edge e is part of a path that informed some other clusters about $\mathcal{C}$, they might merge with $\mathcal{C}$ or some other marked cluster. In either case, e is included in at most one additional cluster. As all of those will merge to the same cluster, we have that e is used by at most $12d^3 + 1 \leq 13d^3$ additional clusters. By induction, there are at most $i \cdot 13d^3 + 13d^3 = (i+1) \cdot 13d^3$ many spanning trees that include a given edge.

Coloring low-degree old clusters that remain. Finally, we are left with only low-degree clusters, as we have included all high-degree clusters in a new cluster. This means that all remaining clusters have at most $2d$ neighboring clusters. We can color these cluster using $O(d^2)$ colors by applying Linial's algorithm [22] which runs in $O(\log^* S)$ rounds of CONGEST on top of the cluster graph, that is, in $2^{O(\sqrt{\log N})} \log^* S$ rounds of the base graph. As we use different colors for each phase, we get a total of $\sqrt{\log N} \cdot O(d)^2 = 2^{O(\sqrt{\log N})}$ colors. ◀

▶ Remark 6. Even though edges can be part of up to $2^{O(\sqrt{\log N})}$ many clusters, per color class they can be included in at most one cluster. This is because otherwise we would have two clusters with the same color that are at distance less than k.

3 Implications on MIS

In this section we present our improved algorithm for computing maximal independent set in the CONGEST model. In particular, we prove the following:

▶ **Theorem 7.** *There is a randomized distributed algorithm, with $O(\log n)$-bit messages, that computes an MIS in $O(\log \Delta \cdot \sqrt{\log \log n}) + 2^{O(\sqrt{\log \log n})}$ rounds, w.h.p.*

We will use the following results about Ghaffari's algorithm for computing a (maximal) independent set [19].

▶ **Theorem 8** ([19])**.** *For each node v, the probability that v has not made its decision within the first $O(\log \deg(v) + \log 1/\epsilon)$ rounds, where $\deg(v)$ denotes v's degree at the start of the algorithm, is at most ϵ.*

▶ **Lemma 9** ([19])**.** *Let B be the set of nodes remaining undecided after $\Theta(\log \Delta)$ rounds. Then, with high probability, we have the following properties:*

(P1) *There is no (G^4)-independent (G^9)-connected subset $S \subseteq B$ s.t. $|S| \geq \log_\Delta n$. This means that S is an independent set in G^4 and induces a connected subgraph in G^9.*

(P2) *All connected components of $G[B]$, that is the subgraph of G induced by nodes in B, have each at most $O(\log_\Delta n \cdot \Delta^4)$ nodes.*

The statement of Lemma 9 is known as a *shattering* guarantee, which is used in various (distributed) algorithms, see e.g. [3,12,13]. Intuitively, this means that after $O(\log \Delta)$ rounds of the algorithm, the components induced by undecided nodes are "small", or more precisely in this case: they do not contain a large 5-independent set. If we allowed for messages of unbounded size, we could just think of the remaining components as graphs of size $O(\log n)$, and use traditional algorithms to solve the problem. However, as we restrict messages to $O(\log n)$-bits, we will need some additional ideas.

We will also use the following ruling set algorithm of Ghaffari [20]:

▶ **Lemma 10** ([20])**.** *There is a randomized distributed algorithm in the* CONGEST *model that, in any network $H = (V, E)$ with at most n vertices, and for any $B' \subset V$ and for any integer $k \geq 1$, with high probability, computes a $(k, 10k^2 \log \log n)$ ruling set $B^* \subseteq B'$, with respect to distances in H, in $O(k^2 \log \log n)$ rounds.*

Algorithm Outline. Combining these results, we can obtain the following: First, we run the algorithm of Ghaffari [19] for $O(\log \Delta)$ rounds, which results in a state described by Lemma 9. Then, we compute a $(5, O(\log \log n)$ ruling set of each remaining component, using Lemma 10. This ruling set induces a clustering, where each node is vertex is clustered to its closest node from the ruling set. By property (P1) of Lemma 9, this yields $N = O(\log n)$ clusters per component of the remaining graph. Let us call one such cluster a *meta-node*, and note that it has diameter $r = O(\log \log n)$. For the remainder of this section, let H be the graph, where the vertex set are all meta-nodes and where two clusters are connected if they contain two adjacent nodes. Then, we compute a network decomposition of H into *super-clusters*. Going through the color classes of this decomposition, one by one, we compute an MIS of each super-cluster. We use the fact that these are graphs of low-diameter to amplify the success probability of a randomized algorithm.

The main challenge will be to compute a suitable network decomposition of H. In particular, we aim to compute a network decomposition using as few colors as possible. In Lemma 12 we obtain such a decomposition, enabling us to prove Theorem 11. We strengthen this result in Lemma 13, using a randomized approach, to get a decomposition that will be sufficient to prove Theorem 7.

3.1 First Approach: Slower but Simpler

In this section, we prove the following Theorem. While it give a slower runtime than Theorem 7, it is still faster than previous algorithms.

▶ **Theorem 11.** *There is a randomized distributed algorithm, with $O(\log n)$-bit messages, that computes an MIS in $O(\log \Delta \cdot \log\log n) + 2^{O(\sqrt{\log\log n})}$ rounds, w.h.p.*

To prove Theorem 11, we use the following algorithm for network decomposition:

▶ **Lemma 12.** *Let H be the $N = O(\log n)$-node graph as described in the outline. There is a deterministic distributed algorithm, with $O(\log n)$-bit messages, that computes a $(O(\log\log n),$ $2^{O(\sqrt{\log\log n})})$ network decomposition of H into* super-clusters *in $2^{O(\sqrt{\log\log n})}$ rounds of communication on G.*

Proof. We will first argue that we can compute a network decomposition of H^k. Then we refine this decomposition into a decomposition of H, while reducing the number of colors used. We will do so by using a *ball growing* process, inspired by [4, 5, 23]: Here, a ball is just a set of vertices with low diameter. Starting from balls being clusters of one color, we grow each of them hop by hop in H until it contains enough meta-nodes. We use the fact that we have a decomposition of H^K (for K large enough) to argue that different clusters can operate independently.

Intermediate Network Decomposition. First, we will compute a network decomposition of H^K for $K = \Theta(\log\log n)$. In G, every meta-node of H is a cluster of diameter $O(\log\log n)$, so we compute such a network decomposition of H^K by computing a network decomposition of G^k for $k = \Theta((\log\log n)^2)$: Using the initial partition as a starting point, we get a $(2^{O(\sqrt{\log\log n})}, 2^{O(\sqrt{\log\log n})})$ network decomposition of G^k by Remark 5. Note that by design of the algorithm, all nodes of such an initial cluster will end up in the same cluster as well. As these initial clusters have diameter $O(\log\log n)$ and we set $k = \Theta((\log\log n)^2)$, two clusters are at distance $\Theta(\log\log n)$ in H.

Now we have an intermediate $(2^{O(\sqrt{\log N})}, 2^{O(\sqrt{\log N})})$ network decomposition of H^K. That is, every two meta-nodes from different clusters of the same color have distance at least $K = O(\log N)$ in H. We can simulate one round of communication within clusters of H in $2^{O(\sqrt{\log n})}$ rounds in G.

One Step of Ball Growing. The next step is to refine this intermediate decomposition to compute a new decomposition of H with the properties from Lemma 12. To do so we use the following ball growing process: In each step, we add some meta-nodes to a new super-cluster, while deactivating another set of meta-nodes. Initially, all meta-nodes are active.

More precisely, the ith step is as follows: Starting from all clusters of color i, we initiate a ball growing process. Note that we only consider meta-nodes that are still active and not yet part of a super cluster. We call a meta-node of H a *boundary* for this ball if at least one of its neighbors is in a different ball. We call a ball *good* if there are less boundary than non-boundary noes.

Initially, a ball is a cluster of color i (or rather its remaining meta-nodes). If the ball is not good, we grow it by one hop in H. We can do this along edges of H, which are also edges in G. In that case, by definition of a good ball, this ball grows by at least a 2 factor in terms of its number of meta-nodes. We repeat this until we reach a good ball. That happens within $\log N$ steps of growth as otherwise the ball would have more than $2^{\log N} = N$ meta-nodes of G, which is not possible. Notice that each step of growth can be performed in $2^{O(\sqrt{\log N})}$ rounds: We aggregate the number of boundary and non-boundary meta-nodes at the center, which then decides whether to stop or continue the process. Once a ball is good, we deactivate its boundary meta-nodes for this phase. The non-boundary meta-nodes of each ball are joined together as one super-cluster of the output-decomposition. In each step of the ball growing, the radius of these super-clusters increases by at most one and thus stays $2^{O(\sqrt{\log N})}$ (which is also true in G, as every meta-node has radius $O(\log N)$). Additionally, balls can grow along each edge at most once, meaning that every edge gets included in at most one additional super-cluster on top of the previous clusters it was included in. Together with the diameter staying $2^{O(\sqrt{\log N})}$, this ensures property (B). We note that the balls that start from different clusters of color i in the intermediate network decomposition can grow simultaneously. They will never reach each other, as originally they were separated by at least $\Omega(\log N)$ hops in H and each ball grows at most $\log N$ hops.

The Full Algorithm. We perform $\log N$ phases: In each phase, we perform $2^{O(\sqrt{\log N})}$ steps of ball growing, one step for each color class of the intermediate network decomposition. Once a phase is finished, we reactive all unclustered meta-nodes and move on to the next phase. Notice that in each phase, at least half of the remaining meta-nodes join a new super-cluster: we only deactivate boundary-nodes and further only do so, whenever we add at least as many nodes to a new super cluster. Thus after $\log N$ phases, the graph must be empty. In total, we spend $\log N \cdot 2^{O(\sqrt{\log N})} \cdot 2^{O(\sqrt{\log N})} = 2^{O(\sqrt{\log N})}$ rounds. As we always deactivate the boundary nodes, super clusters are non-adjacent, which shows property (A). For the number of colors, we use only one color per phase, and as there are $\log N = O(\log\log n)$ phases, we use as many colors. ◀

We can now use this decomposition of G, to compute a maximal independent set:

Proof of Theorem 11. As a first step, we compute H as described before, by running Ghaffari's algorithm [19] for $O(\log \Delta)$ rounds, and computing a clustering in the remaining parts of the graph. Then, we find a $(O(\log\log n), 2^{O(\sqrt{\log\log n})})$ decomposition of H, using Lemma 12.

For computing the MIS we proceed as follows: We work through each of the $O(\log\log n)$ color classes, spending $O(\log \Delta) + 2^{O(\sqrt{\log\log n})}$ rounds per color class. In one color class, we can find one MIS per super cluster, as super clusters of the same color are non-adjacent.

In every step, all nodes of the active super clusters execute $O(\log n)$ parallel executions of the algorithm of Ghaffari [19], as reviewed in Theorem 8. This can be done without any overhead, as every single execution only uses one-bit messages. This super cluster contains $O(\Delta^4 \log n)$ regular nodes, by property (P1) of Lemma 9. Running this algorithm for $O(\log(\Delta^4 \log n)) = O(\log \Delta + \log\log n)$ rounds, we find an MIS with probability at least $1 - 1/\text{poly}(\Delta^4 \log n)$. Since all $O(\log n)$ parallel executions are independent, the probability that none of them succeeds is at most $1/\text{poly}(n)$.

Now we just need to find a run that was successful. For this we use the network decomposition we obtained from Lemma 12. First, each node v performs a local check for all runs, by making sure that either v is in the MIS and none of its neighbors is, or that v is not in the MIS, but at least one of its neighbors is. This can again be done with just

one-bit messages. Then, we can convergecast these local checks towards the cluster centers in $2^{O(\sqrt{\log\log n})}$ rounds. These centers can pick the first successful run and inform all nodes of their cluster in $2^{O(\sqrt{\log\log n})}$ rounds. We remove all nodes that are in the MIS of the super cluster, together with all their neighbors (in the base graph). After this, we move on to the next color.

In total, we spend $O(\log\log n)\cdot(O(\log\Delta+\log\log n)+2^{O(\sqrt{\log\log n})} = O(\log\Delta\cdot\log\log n)+2^{O(\sqrt{\log\log n})})$ rounds and find an MIS with high probability. ◀

3.2 Second Approach: Faster

To improve the runtime compared to Theorem 11 we need to obtain a network decomposition using fewer colors. Instead of a sequential ball growing process, we will perform a randomized *ball carving*, similar to Elkin and Neiman [18]. As this decomposition will be used on small components of the graph, we need to ensure that we still succeed with probability $1 - 1/\text{poly}(n)$ even on components with much less than n vertices. We will use a similar idea as in the proof of Theorem 11, namely that we run many random processes in parallel and use a previously computed network decomposition to find a run that was successful. However, defining the right measure of success and identifying a successful run both require much more care than in algorithm for MIS. The resulting algorithm is formalized in the following Lemma:

▶ **Lemma 13.** *Let H be the $N = O(\log n)$-node graph as described in the algorithm outline. There is a randomized distributed algorithm, with $O(\log n)$ bit messages, that computes a strong diameter $(O(\sqrt{\log\log n}), 2^{O(\sqrt{\log\log n})})$ network decomposition of H in $2^{O(\sqrt{\log\log n})}$ rounds, with probability $1 - 1/\text{poly}(n)$.*

Below, we provide a proof outline, for the complete proof, see the full version of this paper.

Proof Sketch. The general idea is the same as in the proof of Lemma 12: As we can compute a network decomposition of H^k by Theorem 4, we want to use the fact that clusters are separated by k hops, to get a decomposition of H which uses fewer colors. Instead of starting from the initial network decomposition, we restart from scratch, only using the initial network decomposition for amplifying success probabilities.

Let us first quickly recap what *ball carving* is and how it is used in constructing network decompositions: We randomly select a number balls, where each ball B is a set of nodes with low diameter. We call a node a *boundary* of a ball B if it has at least one neighbor that is not in B. For every ball B, we define one cluster C containing all non-boundary nodes of B and remove all such clusters C from the graph. This concludes one step of ball carving.

To create a network decomposition we apply this process recursively on the remaining graph, until it is empty. As we ignore boundary-nodes, the clusters formed in every step are non-adjacent, which means we can use a single color for each recursive step. In order to obtain good bounds for the resulting network decomposition, we need the selected balls to have the following two properties: (A) their diameter is low, and (B) the number of boundary nodes is be small. Intuitively, property (A) means that the resulting decomposition has low diameter, while property (B) means that it uses a small number of colors. We call a ball carving *successful* if properties (A) and (B) are fulfilled.

Given a suitable algorithm for selecting balls, see e.g. [18, 23], this process can be used to compute a network decomposition with much better bounds than what we can compute deterministically. However, the success probability of such selection algorithms is too low in our setting: We only have $N = O(\log n)$ nodes, which means that the success probability is just $1 - 1/\text{poly}(N) = 1 - 1/\text{poly}(\log n)$.

We use the same idea as in the proof of Theorem 11 to amplify this probability: Instead of executing a ball carving on all nodes, we only consider one color class of the initial network decomposition at a time. Next, we execute $O(\log n)$ many parallel attempts of ball carving. This means that with probability $1 - 1/\text{poly}(n)$ at least one attempt was successful. To find such a successful attempt, clusters of the same color can operate independently, as they are separated by k hops. To decide if an attempt was successful, we compute the number of boundary and non-boundary nodes for each ball and aggregate all these numbers at the center node of each cluster. Additionally, we aggregate the maximum diameter amongst all balls. Based on this information, the centers can decide if a attempt was successful. By performing this check for all executions in parallel, we can thus find a successful attempt. Then, we move on to the next color class. ◀

As in the proof of Theorem 11, we can now use this network decomposition to compute a maximal independent set. Since the two proofs are identical, we provide a brief outline:

Proof Sketch of Theorem 7. We can use the Algorithm of [19] to compute an independent set, leaving only components of small diameter in the remaining graph. On these remaining components we compute a $(O(\sqrt{\log\log n}), 2^{O(\sqrt{\log\log n})})$ network decomposition by applying Lemma 13. Then we go through all colors of this network decomposition one by one. Per color class we spend $O(\log\Delta + 2^{O(\sqrt{\log\log n})})$ rounds: We run $O(\log n)$ parallel randomized MIS algorithms for $O(\log\Delta)$ rounds, and use the fact that clusters have radius $2^{O(\sqrt{\log\log n})}$ to pick a successful run in time proportional to this radius. Then we move on to the next color class, removing all nodes that have a neighbor in the computed independent set. In total, this takes $O(\log\Delta) + \sqrt{\log\log n} \cdot 2^{O(\sqrt{\log\log n})}$ rounds to compute an MIS. ◀

References

1. Yehuda Afek and Moty Ricklin. Sparser: A paradigm for running distributed algorithms. *Journal of Algorithms*, 14(2):316–328, 1993.
2. Noga Alon, László Babai, and Alon Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *Journal of algorithms*, 7(4):567–583, 1986.
3. Noga Alon, Ronitt Rubinfeld, Shai Vardi, and Ning Xie. Space-efficient local computation algorithms. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pages 1132–1139. Society for Industrial and Applied Mathematics, 2012.
4. B Awerbuch and D Peleg. Sparse partitions. In *Foundations of Computer Science, 1990. Proceedings., 31st Annual Symposium on*, pages 503–513. IEEE, 1990.
5. Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM (JACM)*, 32(4):804–823, 1985.
6. Baruch Awerbuch. Efficient broadcast and light-weighted spanners. *manuscript*, 1992.
7. Baruch Awerbuch, Shay Kutten, and David Peleg. Online load balancing in a distributed network. In *Proc. 24th ACM Symp. on Theory of Comput*, pages 571–580, 1992.
8. Baruch Awerbuch, Michael Luby, Andrew V Goldberg, and Serge A Plotkin. Network decomposition and locality in distributed computation. In *Foundations of Computer Science, 1989., 30th Annual Symposium on*, pages 364–369. IEEE, 1989.
9. Baruch Awerbuch and David Peleg. Routing with polynomial communication-space srade-ff. *SIAM Journal on Discrete Mathematics*, 5(2):151–162, 1992.
10. Leonid Barenboim. On the locality of some NP-complete problems. In *International Colloquium on Automata, Languages, and Programming*, pages 403–415. Springer, 2012.
11. Leonid Barenboim, Michael Elkin, and Cyril Gavoille. A fast network-decomposition algorithm and its applications to constant-time distributed computation. *Theoretical Computer Science*, 751:2–23, 2018.

12 Leonid Barenboim, Michael Elkin, Seth Pettie, and Johannes Schneider. The locality of distributed symmetry breaking. *Journal of the ACM (JACM)*, 63(3):20, 2016.

13 József Beck. An algorithmic approach to the Lovász local lemma. I. *Random Structures & Algorithms*, 2(4):343–365, 1991.

14 Guy E Blelloch, Anupam Gupta, Ioannis Koutis, Gary L Miller, Richard Peng, and Kanat Tangwongsan. Nearly-linear work parallel sdd solvers, low-diameter decomposition, and low-stretch subgraphs. *Theory of Computing Systems*, 55(3):521–554, 2014.

15 Janosch Deurer, Fabian Kuhn, and Yannic Maus. Deterministic Distributed Dominating Set Approximation in the CONGEST Model. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, page to appear. ACM, 2019.

16 Devdatt Dubhashi, Alessandro Mei, Alessandro Panconesi, Jaikumar Radhakrishnan, and Aravind Srinivasan. Fast distributed algorithms for (weakly) connected dominating sets and linear-size skeletons. *Journal of Computer and System Sciences*, 71(4):467–479, 2005.

17 Michael Elkin. A faster distributed protocol for constructing a minimum spanning tree. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 359–368. Society for Industrial and Applied Mathematics, 2004.

18 Michael Elkin and Ofer Neiman. Distributed strong diameter network decomposition. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, pages 211–216. ACM, 2016.

19 Mohsen Ghaffari. An improved distributed algorithm for maximal independent set. In *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 270–277. Society for Industrial and Applied Mathematics, 2016.

20 Mohsen Ghaffari. Distributed Maximal Independent Set using Small Messages. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 805–820. SIAM, 2019.

21 Mohsen Ghaffari and Fabian Kuhn. Derandomizing distributed algorithms with small messages: Spanners and dominating set. In *32nd International Symposium on Distributed Computing (DISC 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

22 Nathan Linial. Distributive graph algorithms global solutions from local data. In *Foundations of Computer Science, 1987., 28th Annual Symposium on*, pages 331–335. IEEE, 1987.

23 Nathan Linial and Michael Saks. Low diameter graph decompositions. *Combinatorica*, 13(4):441–454, 1993.

24 Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM journal on computing*, 15(4):1036–1053, 1986.

25 Gary L Miller, Richard Peng, and Shen Chen Xu. Parallel graph decompositions using random shifts. In *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*, pages 196–203. ACM, 2013.

26 Alessandro Panconesi and Aravind Srinivasan. Improved distributed algorithms for coloring and network decomposition problems. In *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 581–592. ACM, 1992.

27 David Peleg. Distributed computing. *SIAM Monographs on discrete mathematics and applications*, 5, 2000.

On Bioelectric Algorithms

Seth Gilbert
National University of Singapore, Singapore
seth.gilbert@comp.nus.edu.sg

James Maguire
Georgetown University, Washington, DC, USA
jrm346@georgetown.edu

Calvin Newport
Georgetown University, Washington, DC, USA
cnewport@cs.georgetown.edu

Abstract

Cellular bioelectricity describes the biological phenomenon in which cells in living tissue generate and maintain patterns of voltage gradients across their membranes induced by differing concentrations of charged ions. A growing body of research suggests that bioelectric patterns represent an ancient system that plays a key role in guiding many important developmental processes including tissue regeneration, tumor suppression, and embryogenesis. This paper applies techniques from distributed algorithm theory to help better understand how cells work together to form these patterns. To do so, we present the cellular bioelectric model (CBM), a new computational model that captures the primary capabilities and constraints of bioelectric interactions between cells and their environment. We use this model to investigate several important topics from the relevant biology research literature. We begin with symmetry breaking, analyzing a simple cell definition that when combined in single hop or multihop topologies, efficiently solves leader election and the maximal independent set problem, respectively – indicating that these classical symmetry breaking tasks are well-matched to bioelectric mechanisms. We then turn our attention to the information processing ability of bioelectric cells, exploring upper and lower bounds for approximate solutions to threshold and majority detection, and then proving that these systems are in fact Turing complete – resolving an open question about the computational power of bioelectric interactions.

2012 ACM Subject Classification Applied computing → Biological networks; Theory of computation → Distributed algorithms

Keywords and phrases biological distributed algorithms, bioelectric networks, natural algorithms

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.19

Related Version Full version on arxiv at: `https://arxiv.org/abs/1809.10046`.

Funding *Seth Gilbert*: Supported, in part, by the MOE Tier 2 project "Beyond Worst-Case Analysis: A Tale of Distributed Algorithms" [MOE2018-T2-1-160].
James Maguire: Supported in part by NSF award CCF-1649484.
Calvin Newport: Supported in part by NSF awards CCF-1733842 and CCF-1649484.

1 Introduction & Related Work

An exciting emerging field in cellular biology is the study of *bioelectricity* [17, 25, 22]. This paper applies techniques from distributed algorithm theory to help advance these efforts.

Bioelectricity describes the patterns of voltage differentials caused by differing concentrations of charged ions inside and outside of a cell's plasma membrane. Compelling new lab research, influenced by computer science's use of abstractions, is revealing that these bioelectric patterns can in some cases play the role of high-level programming languages, providing a "biocode" that can specify goal states for cellular development that are then implemented by complex lower-level processes (see [25] for a recent survey).

© Seth Gilbert, James Maguire, and Calvin Newport;
licensed under Creative Commons License CC-BY

In this paradigm, altering a bioelectric pattern – e.g., using interventions such as chemical blockers that modify ion flux, or inserting membrane channels – is like altering the source code of a computer program, providing a mechanism for controlling how an organism develops. The ability to manipulate these processes at a high level of abstraction enables potentially massive breakthroughs in many different important areas, including organ and limb regeneration, tumor suppression, and powerful new forms of synthetic biology.

1.1 Opportunity

The key to unlocking the power of bioelectricity is understanding how the underlying *bioelectric networks* (BENs) interact to form patterns and process environmental input. To date, biologists have primarily studied these questions by describing specific BEN configurations as a system of differential equations, and then studying their behavior using analytical simulation. This provides only observational – *not expository* – insight into bioelectricity dynamics.

In this paper, we explore another investigatory approach that can yield powerful new understanding: the biological algorithms approach [30, 29]. By treating a given BEN configuration as a distributed algorithm running in a well-defined distributed system model, we can apply the tools of distributed algorithms to prove results about a network's behavior, identify network designs that solve specified problems, produce lower bounds and impossibility results, and even assess the general computational power of the setting in question.

To do so, we begin in Section 2 by describing and motivating the *cellular bioloelectric model* (CBM), a new computational model, designed in consultation with biologists who directly study these phenomena, that abstracts important capabilities and constraints of real world cellular bioelectrical networks. This model assumes a collection of *cells*, which are connected in a network topology that describes which cell pairs can directly interact (e.g., through ligand signaling). To simplify the model, time proceeds in synchronous rounds. The state of each cell at the beginning of a round is captured by a single value that describes the voltage *potential* across its plasma membrane. A *gradient* parameter captures the rate at which this potential increases or decreases toward an equilibrium due to ion flux through ion channels in its membrane.

Cells can communicate and compute through *bioelectric events*, in which a cell can induce a sudden increase or decrease to its potential (e.g., by pumping ions in/out, or opening/closing ion gates), and release aionic ligand molecules that can induce a sudden potential changes in its neighboring cells in the network topology. For each cell, and each bioelectrical event, a probability function specific to that event maps the cell's current potential to the probability of the event firing. To maintain biological plausibility, our model requires that these probability function are monotonic, and allows each cell definition to include only a constant number of distinct bioelectric events.

Though the core computational process in the CBM – the *cell* – is quite simple and restricted, we are able to show that they are well-suited to exactly the types of distributed computational tasks that researchers now attribute to bioelectric behavior. Below we summarize our results and emphasize the concrete connections they form to active areas of biological inquiry.

1.2 Our Results: Symmetry Breaking

One of the key open problems in cellular bioelectrics is understanding the stochastic processes that allow otherwise identical cells to distinguish themselves into set patterns. We study these symmetry breaking tasks in Section 3, focusing in particular on the `KnockBack` cell

definition (see Section 3.1). This definition captures one of the simplest possible symmetry breaking strategies. Cells start with a low potential that gradually increases toward a higher equilibrium. As a cell's potential increases, it passes through a *competition* range in which, with constant probability, it fires a bioelectric event that bumps up its potential and emits a ligand that will reduce the potential of nearby cells. If it makes it through the competition range, its potential is high enough that the cell begins firing with probability 1 until its reaches a threshold after which it can begin a morphological transformation into a leader.

Though simple, `KnockBack` turns out to be an effective symmetry breaker. In Section 3.2, we study this strategy in a single hop (i.e., fully connected) network topology. We prove that not only does it safely elect a single cell to be leader, it does so in only $O(\log(n/\epsilon))$ rounds, with probability at least $1-\epsilon$, where n is the network size. For high probability (i.e., $\epsilon < 1/n$), this bound is *faster* than the $O(\log^2 n)$-round algorithm from a recent study of symmetry breaking with constant-size state machines [21]. It also matches the optimal $\Theta(\log n)$ bound on leader election with unrestricted state machines under the comparable network assumptions of a shared communication channel and collision detection [31].

In Section 3.3, we turn our attention to the behavior of `KnockBack` in connected multihop networks that satisfy the natural unit ball graph constraints [23] (which requires the topology to be compatible with the embedding of the cells in a reasonable metric space). In this setting, we consider the *maximal independent set* (MIS) problem, in which: (1) every cell must either become a leader or neighbor a leader; (2) no two neighbors are leaders. Our consideration of the MIS problem is not arbitrary. A 2011 paper appearing in the journal *Science* [4] conjectures that nervous system development in flies solves the MIS problem on a layer of epithelial cells to evenly spread out sensory bristles, motivating the investigation of biologically plausible strategies for solving this classical problem (c.f., [2, 33]).

We show, perhaps surprisingly, that the simple `KnockBack` strategy turns out to provide an effective solution to the MIS problem as well. In more detail, we prove that with high probability in the network size n, it establishes a valid MIS in at most $O(\text{polylog}(\Delta)\log n)$ rounds, where Δ is the maximum degree in the network (which in many biological settings, such as in [4], is likely a small constant).

Equally important for the study of bioelectrics, we show this strategy to be self-stabilizing. Even if you start each cell at an arbitrary initial potential, the system will efficiently stabilize to a valid MIS. The strategy is unique in that it requires only a single constant probability value in its definition, as opposed to the $\log n$ distinct probabilities used in most existing efficient solutions, including those proposed in existing biological distributed algorithm papers [4, 2, 33].

Given these powerful properties of the `KnockBack` strategy, plus a simplicity in design that makes it an easy target for natural selection to identify, we argue that it represents a reasonable (testable) hypothesis that bioelectric mechanisms might be drive these symmetry breaking tasks in real cellular systems.

1.3 Our Results: Information Processing

Another previously mentioned key open problem in cellular bioelectrics is understanding the capacity of cells to process information using bioelectric interactions. One conjecture is that simple interactions of the type captured in the CBM are not capable of much more than simple pattern generation (e.g., generating an MIS with `KnockBack` cells). A competing conjecture is that these interactions are actually capable of performing a wide variety of non-trivial computation.

In this paper, we use the CBM to provide support for the latter view of biological reality. We begin in Section 4 by studying *input type* computation, a simple form of information processing also studied in the biologically-plausible population protocol and chemical reaction network models (see model comparison below). In input type computation, the goal is to compute an output based on the *number* of cells in the system of one or more designated types. Two classical problems of this type are *threshold detection* [5], which computes whether the number of *sick* cells in the system is beyond a fixed threshold k, and *majority detection* [7], which computes whether there are more A cells than B cells in the system.

We study threshold detection in Section 4.1. For small thresholds, we present a simple cell that solves the problem exactly with no error.[1] For larger thresholds, we present a cell definition that for any error ϵ, correctly detects that the threshold is exceeded if the count n is greater than $k\tau$, and correctly detects that it is not exceeded if $n < k/\tau$, for $\tau = O(\log(1/\epsilon))$. We conclude by proving that *any* solution that works for general k values must have a non-zero error probability, regardless of how large we allow τ to grow.

In Section 4.2, we turn our attention to majority detection. We provide symmetric cell definitions for type A and B cells. For any constant error bound $\epsilon > 0$, these cells will correctly detect the majority type with probability $1 - \epsilon$ so long as there is a sufficiently large constant factor more of the majority type (for a constant factor defined relative to $\ln(1/\epsilon)$).

The general threshold detection solution is straightforward: cells send a ligand with probability $1/k$, and associate any received ligands with an exceeded threshold. The majority detection solution has cells increase the firing probability of a bioelectric event from a small lower bound to a constant as their potential increases towards equilibrium: whichever cell type fires first is assumed to be the majority type. In both cases, more refined probabilistic analysis would likely lead to tighter bounds, but the solutions and lower bound in Section 4 are sufficient to support the conjecture that bioelectric interactions can approximate standard input type computations (albeit it only probabilistically).

Finally, in Section 5 we consider a more general form of information processing, in which the input value to be processed in a given execution is encoded in the initial value of one or more designated input cells (for some encoding scheme specified by the designer of the cellular system). Understanding the set of functions that can be computed by such systems provides insights into the computational power of bioelectrics. With this motivation in mind, we prove, perhaps surprisingly, that bioelectric cells are Turing Complete. In slightly more detail, we prove that for any deterministic Turing machine (TM) M, there exists a finite collection of cells including a designated *input cell*, connected in a single hop network, such that for any TM input w, if you set the initial potential value of the input cell to a proper encoding of w, the system will correctly simulate M on w. Of course, one of the TMs that can be simulated is a universal TM, indicating the existence of a computationally universal collection of bioelectric cells.

1.4 Comparison to Existing Models

Generally speaking, in studying the intersection of biology and algorithms there are two main types of computational models used: those with *bio-plausible computation* and those with *bio-plausible constraints*. The first category describes models in which the actual method of computation is motivated by a specific biological context. Algorithms in these models

[1] In this context, “small” means that k is smaller than the maximum number of different ligand counts a cell can can distinguish, allowing the cell to directly count the sick cells (see the definition of *binding bound* from Section 2).

cannot simply be described in standard pseudocode or state machine descriptions. They must instead be specified in terms of the particular bio-plausible computation method captured by the model. Well-known models of this type include neural networks [32, 36, 27, 26], chemical reaction networks [37, 12, 11], and population protocols [5, 8, 7, 9, 6, 10] (which are computationally equivalent to certain types of chemical reaction networks).

The other type of model used to study biological algorithms are those with bio-plausible constraints. These models describe computation with the same standard discrete state machine formalisms assumed in digital computers. They constrain algorithms, however, by adding biologically-motivated limits on parameters such as memory size, the message alphabets used for communication, and the behavior of the communication channels. Well-known models of this type includes the ANTS model [19, 24, 13, 20, 35], the stone age computing model [18], and the beeping model [16, 15, 28, 14, 1, 34, 3, 21].

Both models are useful for applying algorithmic tools to understanding biological systems. The bio-plausible computation models focus more on understanding the low level processes behind particular behaviors, while the bio-plausible constraints models focus more on identifying general distributed strategies, and understanding the minimum resources/assumptions required for useful distributed coordination.

The CBM is most accurately categorized as a bio-plausible computation model. Existing studies of the stone age and beeps computing models already shed light on what can be computed by collections of simple state machines with basic signaling capabilities. The goal here is to understand what can be computed with the *specific* bioelectric mechanisms implemented in living tissue. This goal is important as our work is designed to be relevant to system biologists that are studying and manipulating these specific mechanisms.

1.5 Cells vs. Neurons

There are interesting connections between the CBM and artificial neural network models. The action potential that drives neural computation is itself a bioelectric mechanism. Indeed, many of the basic artificial neural network models can be implemented as special case of our general CBM. The recent work in bioelectricity that motivates the CBM, however, deals with bioelectric activity outside of the neural context, which changes the relevant challenges. In neural networks, for example, the "algorithm designer" gets to carefully construct the network topology and precisely calibrate each cell (i.e., determine their exact connection weights). In the non-neural contexts that motivate this work, by contrast, the network topologies are either simple (single hop) or *a priori* unknown to the computing cells (an arbitrary multihop graph), and because pattern formation is a key behavior in this context, the focus is often on initially identical cells that break symmetry stochastically. In other words, though the CBM networks we study use similar underlying chemical mechanisms as neural networks, their behaviors are strongly distinguished.

1.6 Cells $\neq$ State Machines

A key factor differentiating the CBM from existing bio-plausible constraint models is that the cell formalism is computationally incomparable to a traditional state machine. Consider a basic task such as outputting a repeated pattern: `ABCABCABC...`. This is trivial for a discrete state machine: cycle through three states, one for each output symbol. It is not hard to show, however, that this behavior cannot be implemented by a cell in the CBM. The key difficulty is the required monotonicity for firing functions driving bioelectric events (which is an important property of the real biological cells being modelled). A simple argument

establishes that for any cell, there must be at least two symbols $S_1, S_2 \in \{A, B, C\}$, such that whenever S_1 has a non-zero probability of being output, so does S_2 – eliminating the possibility of perfectly repeating pattern. At the same time, we cannot necessarily simulate an arbitrary cell with a finite state automaton either, since each cell stores an analog and unbounded potential value. It is therefore unclear how to use an existing bio-plausible constraint model to directly explore bioelectric dynamics – directly modeling the bioelectric dynamics seems necessary for understanding these systems.

1.7 Why Study This Model?

A shortcoming of the biological algorithms approach is that it can spawn an unlimited number of new models. The difficult question for advancing this field is identifying *which* models are actually worth ongoing examination In defense of the CBM, we note that it was created in response to interactions with biologists who were excited about the potential of bioelectricity and increasingly comfortable borrowing ideas from computer science. The details of the CBM presented here were identified in consultation with these biologists, and the initial problems we study were directly motivated by questions in the existing literature. Even so, we made several modelling decisions that can be questioned (e.g., regarding both fidelity and tractability), and only future attempts to use the CBM to better understand biology will help to resolve those questions. Successful synthesis of algorithm theory and biology is an exceedingly hard endeavor, but we contend that this direction is well-motivated.

2 The Cellular Bioelectric Model

Here we define the cellular bioelectric model (CBM), a synchronous computation model that abstracts the key capabilities and constraints of bioelectric networks.

2.1 Biology Background

A bioelectric network describes the bioelectric properties of a collection of cells in some well-defined space. The key property describing the network is the net difference in charged ion concentration between the inside of each cell and the extracellular environment. There are two main mechanisms by which the voltage across a given cell's plasma membrane can change. The first is charged ions moving in or out of membrane channels driving the cells interior ion concentration toward equilibrium with the outside extracellular environment. The second mechanism is ligand signalling. A given cell's voltage can induce the release of special signalling molecules called ligands into the extracellular environment. These ligands can then bind to receptors on nearby cells, either opening channels in the receiver's membrane or activating ion pumps, rapidly changing the ion concentration of the receiver. The release of ligands by the cell can also cause a sharp change to its own ion concentration through similar mechanisms. These bioelectric events are stochastic in nature with a probability that seems to depend monotonically on a cell's current voltage; e.g., the probability of an event either becomes increasingly more or less likely as the voltage grows.

The below model captures the core properties of these dynamics. The voltage of each cell is captured by a single analog *potential value*, while we capture the passive drive toward equilibrium with both an equilibrium value and a rate at which each cell's potential drives toward that equilbrium. Bioelectric events are described by probability functions that map cellular potential values to the probability of the event firing. Finally, we use a graph to describe the cellular topology, where an edge (u, v) means that the cells corresponding to u and v are within ligand signalling range.

By necessity, this model simplifies the real biology in several important ways. For example, our discrete bioelectric events actually approximate analog non-linear responses to signaling, and likely limit the full range of signalling interactions possible in real systems. In addition, we consider only anionic ligands (no charge), whereas some well-known bioelectric interactions seem to rely on cationic ligands that change the charge of the extracellular environment. Also notable, for the sake of simplicity, we omit the inclusion of *gap junctions*, which are direct channels between cell pairs that can open and close in response to the voltage gradient induced by their endpoints.

2.2 Cells

Fix a non-empty and finite set L containing the *ligands* cells use to drive bioelectrical interactions. We define a *bioelectric event* to be a pair $(f,(\delta,s))$, where $f:\mathbb{R}\rightarrow[0,1]$ is a *firing function* from real numbers to probabilities, and (δ,s) consists of a *potential offset* value $\delta\in\mathbb{R}$, and a *ligand* $s\in L$. We also define a *membrane function* to be a function g from multisets defined over L to real numbers.

Pulling together these pieces, a *cell* in our model is described by a 6-tuple $(q_0,\sigma,\lambda,\omega,g,\mathcal{B})$, where $q_0\in\mathbb{R}$ is the *initial potential* value of the cell, $\sigma\in\mathbb{R}$ is the *equilibrium* potential that the cell will drive its internal potential toward (i.e., through ion flux), $\lambda\in\mathbb{R}^+$ is a non-negative real number describing the *gradient* rate at which the cell's potential moves toward σ, $\omega\in\mathbb{R}$ is the smallest possible potential for the cell, g is a membrane function, and $\mathcal{B}$ is a set of bioelectric events. For a given cell c, we use the notation $c.q_0, c.\sigma, c.\lambda, c.\omega, c.g, c.\mathcal{B}$ to refer to these six elements of the cell's tuple.

2.3 Systems and Executions

A *system* in our model consists of a non-empty set $\mathcal{C}$ of $n=|\mathcal{C}|$ cells, an undirected graph $G=(V,E)$ with $|V|=n$, and a bijection $i:\mathcal{C}\rightarrow V$ assigning cells to graph vertices. For simplicity, in the following we sometimes use the terms *cell* u or *node* u, for some $u\in V$, to refer to the unique cell $c\in\mathcal{C}$ such that $i(c)=u$.

An *execution* proceeds in synchronous rounds that we label $1,2,3,\ldots$. At the beginning of each round r, we define the *configuration* $C_r:\mathcal{C}\rightarrow\mathbb{R}$ as the bijection from cells to their potential values at the beginning of round r. For each $c\in\mathcal{C}$, $C_1(c)=c.q_0$. That is, each cell starts with the initial potential value provided as part of its definition. The configuration for each round $r>1$ will depend on the configuration at the start of round $r-1$, and the (potentially probabilistic) behavior of the cells during round $r-1$.

In more detail, each round $r\geq 1$ proceeds as follows:

1. For each cell $c\in\mathcal{C}$, initialize $p_c\leftarrow C_r(c)$ to c's potential at the start of round r. We will use p_c to track how c's potential value changes during this round. Also initialize multiset $M_c=\emptyset$. We will use M_c to collect ligands sent toward c during this round.
2. For each cell $c\in\mathcal{C}$, and each bioelectric event $(f,(\delta,s))\in c.\mathcal{B}$, this event *fires* with probability $f(C_r(c))$. If the event fires, update $p_c\leftarrow p_c+\delta$ and add a copy of s to multiset $M_{c'}$, for each cell $c'\in\mathcal{C}$ such that $\{i(c),i(c')\}\in E$ (that is, for each cell c' that neighbors c in G).
3. After processing all rules at all cells, the round proceeds by having cells process their incoming ligands. For each cell $c\in\mathcal{C}$, update $p_c\leftarrow p_c+c.g(M_c)$. That is, update the potential change according to c's membrane function applied to its incoming ligands.

4. Finally, we calculate the impact of the gradient driving each cell c's potential toward its equilibrium value. In more detail, let $z = C_r(c) - c.\sigma$. We define the gradient-driven potential change for c in round r, denoted $\lambda_r(c)$, as follows:

$$\lambda_r(c) \leftarrow \begin{cases} -c.\lambda & \text{if } z \geq c.\lambda \\ -z & \text{if } 0 < z < c.\lambda \\ 0 & \text{if } z = 0 \\ z & \text{if } -c.\lambda < z < 0 \\ c.\lambda & \text{if } z \leq -c.\lambda \end{cases}$$

   We add this gradient-induced offset to c's potential: ,$p_c \leftarrow p_c + \lambda_r(c)$.
5. The final step is to the initial potential for $r+1$ for each $c \in \mathcal{C}$, by performing a final check that the potential did not fall below the cell's lower bound in the round: $C_{r+1}(c) \leftarrow \max\{p_c, c.\omega\}$.

2.4 Natural Constraints on Cell Definitions

To maintain biological plausibility, our model includes the following natural constraints on allowable cell definitions:

- *Constraint #1:* Each cell definition includes at most a constant number of bioelectric events.
- *Constraint #2:* Firing functions are monotonic.
- *Constraint #3:* For each membrane function g, there must exist some constant $b > 0$, such that for every possible ligand multiset M, $g(M) = g(\hat{M})$, where $\hat{M}$ is the same as M except every value that appears *more* than b times in M is replaced by exactly b copies of the value in $\hat{M}$. We call the value b the *binding bound* for that cell definition.

2.5 Expression Events & Thresholds

In real biological systems, bioelectric patterns induce morphological changes driven by lower-level processes. To capture this transformation we introduce the notion of *expression events* into our model (named for the idea that bioelectics regulates gene *expression*).

In more detail, some of our problem definitions specify a potential threshold such that if a cell's potential exceeds this threshold, an irreversible morphological transformations begins. This occurs at the beginning of each round, i.e., if a cell begins round r with a potential that exceeds the event threshold, we apply the event. For example, in studying leader election (see Section 3), we assume once a cell passes a given threshold value with its potential it transforms into a *leader*, at which point it stops executing its original definition and transforms neighbors that have potential values below the threshold into *non-leaders*. The specification and motivation for specific expression thresholds are included as part of the problem definitions.

3 Symmetry Breaking

A fundamental task in bioelectric networks is generating non-trivial bioelectric patterns that can then direct cellular development. This requires symmetry breaking. With this in mind, we study the symmetry breaking capabilities of a natural, but surprisingly effective, cell called `KnockBack`. We summarize its ability to elect a leader in single hop networks, and to efficiently generate maximal independent sets in multihop networks.

3.1 The `KnockBack` Cell

We define a `KnockBack` cell as follows:

`KnockBack` cell definition	
$q_0 = 0$	$\mathcal{B} = \{(f, (1/2, m))\}$, where:
$\lambda = 1/2$, $\sigma = 2$, $\omega = -2$	$f(x < 1/2) = 0$
$g(\lvert M\rvert > 0) = -(3/2)$	$f(1/2 \leq x < 1) = 1/2$
$g(\lvert M\rvert = 0) = 0$	$f(x \geq 1) = 1$
leader expression rule threshold: ≥ 2	

The `KnockBack` cell implements a natural symmetry breaking strategy. It is initialized with a low initial value of $q_0 = 0$ that is driven toward the equilibrium of $\sigma = 2$ at a gradient rate of $\lambda = 1/2$. As a cell's potential value passes through the range of $[1/2, 1)$, its single bioelectric event $(f, (1/2, m))$ fires with constant probability. If this event fires, the cell increases its potential by $1/2$ (e.g., by pumping in more ions), and emits the ligand m, which will bind with its neighbors in the topology. If at least one of the cell's neighbor emits the ligand m, then that cell will decrease its potential by $-(3/2)$ (e.g., by pumping out ions).

If a cell makes it to a potential value of 1 or greater, this event starts firing with probability 1. If a cell makes it to potential value of 2 or greater, it executes the *leader expression event*, which makes it a leader, and makes each neighbor below the threshold into non-leaders.

Two neighbors cannot both become leaders because any cell that becomes a leader in some round $r + 1$, must have spent round r at a potential value where it fires its bioelectric event with probability 1. If two neighbors fire this event in r, however, they both have a net decrease in their potential, preventing them from becoming leaders in $r + 1$. The time required for a leader to emerge is more complicated to derive, especially in the multihop context. The intuition behind these analyses, however, is that when multiple nearby cells simultaneously have potential values in the *competition range* of $[1/2, 1)$, it is likely that some will fire their event and some will not, aggregating inequality in their competition status until only a single leader remains.

3.2 Single Hop Leader Election

Consider a single hop (i.e., fully-connected) network consisting of $n > 0$ copies of the `KnockBack` cell defined in Section 3.1. We study the ability of this system to solve the leader election problem, which requires the system to converge to a state in which one cell is a leader and all other cells are non-leaders. We prove that the system never elects more than one leader, and that for any error probability $\epsilon > 0$, with probability at least $1 - \epsilon$ it elects a leader in $O(\log(n/\epsilon))$ rounds. As we detail in Section 1, this round complexity is comparable to the best-known solutions in more powerful computational models. Formally:

▶ **Theorem 1.** *Fix some error bound $\epsilon > 0$ and network of $n \geq 1$ `KnockBack` cells. With probability at least $1 - \epsilon$, a leader is elected within $O(\log(n/\epsilon))$ rounds. There is never more than 1 leader elected.*

The full proof, deferred to the full version of this paper (found on arXiv), tackles the liveness and safety properties separately. The safety property follows directly from the argument summarized above about the impossibiity of two cells making it through the *gateway* potential where both would fire events and knock each other back out of immediate contention for leadership. The liveness argument proves that the set of contenders probabilistically bifurcates over time into two set A and B, where once in B a cell is no longer ever again in contention. We now provide a brief summary of the analysis.

Preliminaries

To understand the leader election process, we must first understand how the potential of a cell evolves. In each round, a cell c changes potential for three reasons: it sends a ligand, it receives a ligand, and the positive gradient, as summarized in this table:

	send	**no send**
receive	$-1/2$	-1
no receive	1	$1/2$

Safety

We now prove that at most one cell becomes leader.

▶ **Lemma 2.** *A single hop network comprised of `KnockBack` cells never elects more than one leader.*

Proof. Assume for contradiction that two different cells c and c' both become a leader during the same round $r > 1$.

Since c and c' first reached potential ≥ 2 in round r, the largest possible increase in potential is 1, and the smallest possible increase in potential is $1/2$, it follows that both must have started round $r-1$ with a potential in $\{1, 3/2\}$.

Therefore, both c and c' sent ligands in round $r-1$, and hence both c and c' *decreased* their potential by $1/2$ during round $r-1$, starting round r with a potential in $\{1/2, 1\}$, contradicting the assumption that both cells are elected leader in r. ◀

Time Complexity

We now show that it does not take too long to elect a leader with reasonable probability. We first identify a set of contenders. Let $p(r)$ be the maximum potential of any cell in round r, and let $A(r)$ be the cells with potential $p(r)$; these are the contenders. Let $B(r)$ be all the other cells with potential $< p(r)$, i.e., the non-contenders. We can show, by a case analysis, that once a cell is no longer contending, it will never contend again:

▶ **Lemma 3.** *If cell c is in $B(r)$ in some round r, then cell c is in $B(r')$ for all $r' \geq r$.*

We say that a round is a *competition round* if there is at least one cell with potential at least $1/2$, and no cell with potential at least 1. In a contention round, there are at least some cells that send ligands with probability $1/2$, and no cell that sends ligands with probability 1. We can show, again by a case analysis, that competition rounds occur frequently:

▶ **Lemma 4.** *Fix some round $r \geq 1$. If r is a competition round then either: $r+2$ is a competition round or a leader is elected by $r+2$.*

Since (by definition) round 2 is a competition round, in fact, every even round will be a competition round. An important property of competition rounds is that with constant probability they reduce the number of cells in A by a constant fraction due to the case in which some cells send a ligand and some do not.

▶ **Lemma 5.** *If r is a competition round and $A(r)$ contains at least 2 cells, then with probability at least $1/12$, the set $A(r+1) \leq (3/4)A(r)$.*

We can now conclude the proof that there is eventually one leader. We know that all the even rounds are competition rounds, and in each even round we reduce the competitor set $A(r)$ by a constant fraction with constant probability, as long as their are at least two competitors. The set $A(r)$ never becomes empty (as there is always some cell with the maximum potential), and never increases in size. Hence by a Chernoff Bound, with probability $1-\epsilon$, within $O(\log(n/\epsilon))$ rounds there have been at least $\log n$ rounds in which $A(r)$ has been successfully reduced by a constant fraction, implying that at this point, the set $A(r)$ contains only one cell. The last remaining competitor becomes leader soon after that occurs.

3.3 Maximal Independent Sets

We now study the behavior of the `KnockBack` cell when executed in a multihop network topology that satisfies the natural *unit ball graph* property (see [23]). We show, perhaps surprisingly, that this simple cell efficiently solves the *maximal independent set* (MIS) problem in this context – providing what is arguably one of the simplest and most biologically-plausible explanations for how interacting cells might generate these useful patterns. Details and proofs can be found in the full version of this paper (posted on arXiv).

Solving the MIS problem requires that the system satisfy the following two properties: (1) *maximality*, every cell is a leader or neighbors a leader; and (2) *independence*, no two neighbors are leaders. We prove that the leaders elected by `KnockBack` in a multihop network always satisfy property 2, and that with high probability in the network size n, property 1 is satisfied in $O(\text{polylog}(\Delta)\log n)$ rounds, where Δ is the maximum degree in the network topology (and in many biological contexts, likely a small constant). We then show that the algorithm still efficiently *stabilizes* to an MIS even if we start cells at arbitrary potential values, an important property for noisy biological contexts.

As we elaborate in Section 1, the simplicity, efficiency, and stabilizing nature of generating MIS's with `KnockBack` leads us to hypothesize that bioelectrics might play a role in the observed generation of MIS patterns in the epithelial cells of flies [4]. The round complexity of our solutions, though not theoretically optimal, is comparable to existing solutions in more powerful computation models. Formally:

▶ **Theorem 6.** *Consider a network of $n \geq 1$ `KnockBack` cells connected in a unit ball graph G with constant doubling dimension and maximum degree Δ. With probability at least $1-1/n$, all cells terminate within $O(\text{polylog}(\Delta)\log(n))$ rounds, with the set of resulting leaders defining an MIS on G.*

Safety

First, we observe that if a cell reaches potential 1.5, then forever thereafter it continues to have high potential, while all of its neighbors remain with negative potential. This immediately implies that two neighbors cannot both be in the MIS. The argument here is nearly identical to Lemma 2.

▶ **Lemma 7.** *Let c and c' be two neighboring cells. It is never the case c and c' both have potential > 1.5.*

Time Complexity

The more interesting task is proving that eventually, every cell or one of its neighbors will enter the MIS, and that this will happen quickly.

A cell is said to be in the MIS if it has potential at least 2. We analyze the behavior of *active* cells, i.e., those that are not in the MIS and that do not have any neighbors in the MIS.

We focus on cells whose potential is a local maximum, i.e., where every neighbor of c has potential no greater than that of c. If a cell is a local maximum, it may still have neighbors of equal potential – these are its competitors for entering the MIS. In fact,if a cell c is a local maximum and, and if cell c has approximately d competitors with equal potential, then it has (approximately) probability $1/d$ of entering the MIS within $O(\log \Delta)$ rounds.

We will want to identify cells that are likely going to enter the MIS quickly, or have a neighbor that is likely to enter the MIS quickly. We define a *quick-entry* cell as follows:

- Cell c is active.
- Cell c is a local maximum.
- Every neighboring competitor of c (with equal potential to c) is also a local maximum.
- If cell c has d neighboring competitors, then each of the neighboring competitors has at most $2d$ neighboring competitors of its own.

We will show that if c is a quick-entry cell, then either it or one of its neighbors will enter the MIS quickly, since each of these $d+1$ cells has (approximately) probability $\geq 1/2d$ of entering the MIS (sidestepping issues of independence, which is the key challenge in proving this lemma).

▶ **Lemma 8.** *Consider the subgraph consisting only of active cells. Let c be a quick-entry cell. Then with probability at least $1/16$, either c or a neighbor of c enters the MIS within $O(\log \Delta)$ rounds.*

Proof (Sketch). Let S be the set consisting of c and its neighbors with the same potential. Let $s = |S|$. Notice every cell in S has at most $2s$ neighboring competitors, and recall that every cell in S is a local maximum.

In every round, we update S as follows: if $c' \in S$ is a cell in S, and if the current round is a competition round for c' in which c' does not send a ligand, then we remove c' from S.

S is the set of cells that remain candidates for entering the MIS, and every cell in S remains a local maximum. All the cells in S will maintain the same potential. Competition rounds are those in which cells in S have potential $1/2$. Cell in S continue entering competition rounds every other round until either S is empty or some cell in S enters the MIS.

A cell in S is a winner if, over $\log(4s)$ competition rounds: (i) it sends in all the competition rounds, and (ii) every one of its neighbors with the same potential, but *not* in S, has at least one competition round in which it does not send. Since each cell has at most $2s$ such neighbors, we can show that the probability that a cell in S wins is at least $1/(8s)$.

We can then analyze the event $W(c')$ that: (i) cell c' is a winner, and (ii) no other cell in S sends in all the competition rounds. These events are disjoint, and the probability of a cell in S sending in all the competition rounds is independent of the behavior of other cells in S. So we can show that for each cell c' in S, this event $W(c')$ occurs with probability at least $1/(16s)$.

This implies that with probability $\geq 1/16$, by the end of the competition rounds, there is exactly one cell c' in S that is a winner, and goes on to enter the MIS in $O(1)$ rounds. ◀

Next, we show that there is always a quick-entry cell no more than $O(\log \Delta)$ hops away:

▶ **Lemma 9.** *Consider the subgraph consisting only of active cells. For every cell c' active in round r, there exists a quick-entry cell c within distance $O(\log \Delta)$.*

Proof (Sketch). The proof of this is constructive, beginning at cell c and moving through the graph until we find a suitable cell not too far from c.

Beginning at c, we repeatedly move to any active cell within distance $(\log(\Delta) + 2)$ that has larger potential. Since potential ranges from -3 to 2 by multiples of $1/2$, within 10 steps this process stops at some c'. All the cells with the same potential as c' within distance $\log(\Delta) + 2$ of c' are local maxima.

Next, we repeat the following: If c' has d neighbors that are competitors (i.e., have the some potential), and if any neighbor of c' that is a competitor has more than $2d$ neighbors that are competitors, then we move to that neighbor. Since the number of neighboring competitors doubles at each step, this terminates within $\log \Delta$ rounds.

The resulting cell is a quick-entry cell, and within distance $O(\log(\Delta))$ of the initial cell c. ◀

Putting together the previous two lemmas, we conclude:

▶ **Lemma 10.** *Given any cell c active in round r, with probability at least $1/16$ there is a cell within distance $O(\log \Delta)$ that enters the MIS within $O(\log \Delta)$ rounds.*

Finally, we leverage the assumption that the underlying graph topology $G = (V, E)$ is a UBG with constant doubling dimension. A graph $G = (V, E)$ is UBG [23] if it satisfies the following two constraints: (1) there exists an embedding of the nodes in V in a metric space such that there is an edge $\{u, v\}$ in E if and only if $dist(u, v) \leq 1$; and (2) the doubling dimension of the metric space, defined as the smallest ρ such that every ball can be covered by at most 2^ρ balls of half its radius, is constant. (In the real-world, where physical cells are embedded in a two or three-dimensional Euclidean space and neighboring cells can interact, the resulting topology is UBG.) UBG graphs provide the following standard property:

▶ **Lemma 11.** *For every independent set I and cell c, there are $O(k^\rho)$ cells in I within distance k of c.*

We can now prove Theorem 6 by arguing that for a cell c, it either enters the MIS or it has a quick-entry cell within distance $O(\log(\Delta))$ that enters the MIS with constant probability. Since there are a bounded number of cells within distance $O(\log(\Delta))$ that can legally enter the MIS (due to the UBG property), we can bound how long until cell c is no longer active.

Stabilization

Throughout the analysis above, we assumed for simplicity that all the cells began with potential precisely zero. However, it turns out that is not in fact necessary. Notably, if the potentials begin too low, e.g., < -3, then eventually the potential climbs into the normal range (due to the gradient effect), unless a neighbor joins the MIS first and preempts it. Alternatively, if potentials begin too high and two neighboring nodes have potential > 1, then they will continue to send in every round and hence eventually one or both will exit the MIS, with their potential dropping below 2. Once safety has been restored, i.e., no neighbors are in the MIS, then the system will stabilize as already described. Nowhere in the analysis did we depend on any special initial conditions or relations between the potentials. Thus we conclude:

▶ **Theorem 12.** *Consider a network of $n \geq 1$ `KnockBack` cells connected in a unit ball graph G with constant doubling dimension and maximum degree Δ. Assume that the cells begin with arbitrary potentials. Then eventually, with probability 1: no two neighboring cells are in the MIS, and every cell is either in the MIS or has a neighbor in the MIS.*

4 Input Type Computation

We now turn our attention to processing information, beginning with a problem studied in bio-inspired chemical reaction networks and population protocols: computation on input type counts. For these problems the *input* is the *a priori* unknown counts of the different cell types in the system. We look at two commonly studied problems: threshold and majority detection, establishing that these problems are tractable in the CBM, but require randomized solutions with non-zero error probabilities. Full details appear in the full version of the paper (posted on arXiv).

4.1 Threshold Detection

The threshold detection problem, which is parameterized with a threshold k, approximation factor τ, and error bound ϵ, and requires a correct answer if the number of sick cells is larger than $\tau \cdot k$, or less than k/τ (see the full version of the paper for the formal definition).

For the sake of completeness, in the full version of this paper we start by describing and analyzing a simple cell definition called `SmallThreshold`(k), that works when the binding bound (see Section 2) is large enough for cells to directly count up to k, trivializing the problem, even for $\epsilon = 0$ and $\tau = 1$. For larger k values, we consider the following more general probabilistic solution:

`GeneralThreshold`(k) cell definition	
$q_0 = 1$	$\mathcal{B} = \{(f, (2, m))\}$, where:
$\lambda = 1$, $\sigma = 0$	$f(x \geq 1) = 1/k$
$g(\lvert M \rvert \geq 1) = 2$	$f(x < 1) = 0$
$g(\lvert M \rvert < 1) = 0$	
event threshold: 2	

The `GeneralThreshold`(k) cell has cells fire a bioelectric event with probability $1/k$. If *any* cell fires, it moves itself past the event threshold, otherwise, the system falls back to a quiescent equilibrium. In the full version of the paper, we show a strict trade-off between the error bound and τ approximation:

▶ **Theorem 13.** *Fix any error bound $\epsilon, 0 < \epsilon < 1$ and threshold $k \geq 1$. Then the `GeneralThreshold`(k) cell definition solves the $(k, 8\ln(1/\epsilon), \epsilon)$-threshold detection problem in one round.*

Another possible improvement would be removing the non-zero error bound (i.e., achieving $\epsilon = 0$), or finding a deterministic solution. We prove such improvements are impossible (see the full version for more details):

▶ **Theorem 14.** *Fix a binding bound $b \geq 1$, threshold range $\tau \geq 1$, and round length $T \geq 1$. There does not exist a cell definition with binding bound b that solves the $(k, \tau, 0)$-threshold detection problem in T rounds for every threshold $k \geq 1$. Fix ϵ, $0 \leq \epsilon < 1/2$. There does not exist a* deterministic *cell definition with binding bound b that solves the (k, τ, ϵ)-threshold detection problem in T rounds for every threshold $k \geq 1$.*

4.2 Majority Detection

Majority detection assumes two cell types: A and B. The goal is to determine which type is more numerous. As with threshold detection, and most existing studies of majority detection in other models (e.g., [7]), we look at approximate solutions that ensure a correct answer only if one count is sufficiently larger than the other. We tackle this challenge with the below cell definition which is parameterized with an upper bound N on the maximum network size and a constant error bound $\epsilon > 0$:

`MajorityA`$(N, \alpha = \lceil 2\ln(2/\epsilon)\rceil)$ cell definition (for type A)	
$q_0 = 0$	$\mathcal{B} = \{(f, (\alpha \log N, m_A))\}$, where:
$\lambda = 1$, $\sigma = 3\alpha \log N$	$f(0 \leq x \leq \alpha \log N) = 2^{-(\log N - \lfloor \frac{x}{\alpha} \rfloor)}$
$g(\lvert M_B\rvert \geq 1) = -2\alpha \log N$	$f(x < 0) = 0$
$g(\lvert M_B\rvert = 0) = 0$	$f(x > \alpha \log N) = 1$
event threshold: $3\alpha \log N$	
(M_B equals the sub-multiset including only ligands of type m_B sent from type B cells.)	

This cell implements a common backoff style strategy, perhaps inspired from radio networks, where nodes fire with increasing probabilities. The first cell type to fire is assumed to be the majority type in the system. In the full version of this paper, we show a trade-off between ϵ and the required size gap between the cell type counts:

▶ **Theorem 15.** *Fix some constant error bound $\epsilon > 0$ and upper bound $N > 1$. Let $\alpha = \lceil 2\ln(2/\epsilon)\rceil$. The `MajorityA`$(N, \alpha)$ and `MajorityB`(N, α) cell definitions, when executed in a system with n_A and n_B type A and type B cells, respectively, where $n_A > n_B \cdot (\alpha 4)/\epsilon$ and $N \geq n_A + n_B$, guarantees with probability at least $1 - \epsilon$: in the first $O(\log n)$ rounds, a type A expression event will occur before any type B event. (The symmetric claim also holds for $n_B > n_A \cdot (\alpha 4)/\epsilon$.)*

5 Turing Completeness

Finally, we consider another natural definition of information processing in which cells compute functions on an input encoded in the potential of a designated *input cell*. This isolates a core question: *What types of computations on cell states can be computed through simple bioelectric interactions?* In the full version of the paper (posted on arXiv), we prove a perhaps surprising answer: Essentially all feasible computations.[2] Formally:

▶ **Theorem 16.** *Fix an arbitrary deterministic TM M. There exists a finite collection of cells defined with respect to M, including a designated* input cell*, such that for every TM input w, if you set the input cell's initial potential value to a specified unary encoding of w, the cells will correctly simulate M on w.*

References

1. Yehuda Afek, Noga Alon, Ziv Bar-Joseph, Alejandro Cornejo, Bernhard Haeupler, and Fabian Kuhn. Beeping a Maximal Independent Set. In *Proceedings of the Symposium on Distributed Computing (DISC)*, 2011.
2. Yehuda Afek, Noga Alon, Ziv Bar-Joseph, Alejandro Cornejo, Bernhard Haeupler, and Fabian Kuhn. Beeping a maximal independent set. *Distributed Computing*, 26(4):195–208, 2013.

[2] Turing completeness is relatively common in restricted state machine models, yet it did not seem *a priori* obvious whether the CBM would satisfy this property as *computation* in the CBM is restricted to simple bioelectric interactions.

3 Yehuda Afek, Noga Alon, Ziv Bar-Joseph, Alejandro Cornejo, Bernhard Haeupler, and Fabian Kuhn. Beeping a maximal independent set. *Distributed Computing*, 26(4):195–208, 2013.

4 Yehuda Afek, Noga Alon, Omer Barad, Eran Hornstein, Naama Barkai, and Ziv Bar-Joseph. A biological solution to a fundamental distributed computing problem. *Science*, 331(6014):183–185, 2011.

5 Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, 18(4):235–253, 2006.

6 Dana Angluin, James Aspnes, and David Eisenstat. Stably computable predicates are semilinear. In *Proceedings of the Symposium on Principles of Distributed Computing (PODC)*, pages 292–299, 2006.

7 Dana Angluin, James Aspnes, and David Eisenstat. A simple population protocol for fast robust approximate majority. *Distributed Computing*, 21(2):87–102, 2008.

8 Dana Angluin, James Aspnes, and David Eisenstat. Fast computation by population protocols with a leader. *Distributed Computing*, 21(3):183–199, 2008.

9 Dana Angluin, James Aspnes, David Eisenstat, and Eric Ruppert. The computational power of population protocols. *Distributed Computing*, 20(4):279–304, 2007.

10 Ioannis Chatzigiannakis and Paul G. Spirakis. The Dynamics of Probabilistic Population Protocols. In *Proceedings of the Symposium on Distributed Computing (DISC)*, 2008.

11 Ho-Lin Chen, Rachel Cummings, David Doty, and David Soloveichik. Speed faults in computation by chemical reaction networks. *Distributed Computing*, 30(5):373–390, 2017.

12 Ho-Lin Chen, David Doty, and David Soloveichik. Deterministic function computation with chemical reaction networks. *Natural computing*, 13(4):517–534, 2014.

13 Alejandro Cornejo, Anna Dornhaus, Nancy Lynch, and Radhika Nagpal. Task allocation in ant colonies. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, pages 46–60. Springer, 2014.

14 Alejandro Cornejo and Fabian Kuhn. Deploying Wireless Networks with Beeps. In *Proceedings of the Symposium on Distributed Computing (DISC)*, 2010.

15 Julius Degesys and Radhika Nagpal. Towards desynchronization of multi-hop topologies. In *Proceedings of the International Conference on Self-Adaptive and Self-Organizing Systems, 2008 (SASO)*, 2008.

16 Julius Degesys, Ian Rose, Ankit Patel, and Radhika Nagpal. DESYNC: self-organizing desynchronization and TDMA on wireless sensor networks. In *Proceedings of the International Conference on Information Processing in Sensor Networks*, 2007.

17 Fallon Durant, Junji Morokuma, Christopher Fields, Katherine Williams, Dany Spencer Adams, and Michael Levin. Long-term, stochastic editing of regenerative anatomy via targeting endogenous bioelectric gradients. *Biophysical journal*, 112(10):2231–2243, 2017.

18 Yuval Emek and Roger Wattenhofer. Stone Age Distributed Computing. In *Proceedings of the Symposium on Principles of Distributed Computing (PODC)*, 2013.

19 Ofer Feinerman, Amos Korman, Zvi Lotker, and Jean-Sébastien Sereni. Collaborative search on the plane without communication. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 77–86. ACM, 2012.

20 Mohsen Ghaffari, Cameron Musco, Tsvetomira Radeva, and Nancy Lynch. Distributed house-hunting in ant colonies. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 57–66. ACM, 2015.

21 Seth Gilbert and Calvin Newport. The computational power of beeps. In *International Symposium on Distributed Computing*, pages 31–46. Springer, 2015.

22 Jessica Hamzelou. Bioelectric tweak makes flatworms grow a head instead of a tail. *New Scientist*, May 2017.

23 Fabian Kuhn, Thomas Moscibroda, and Rogert Wattenhofer. On the locality of bounded growth. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 60–68. ACM, 2005.

24 Christoph Lenzen, Nancy Lynch, Calvin Newport, and Tsvetomira Radeva. Trade-offs between selection complexity and performance when searching the plane without communication. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 252–261. ACM, 2014.

25 Michael Levin and Christopher J Martyniuk. The bioelectric code: An ancient computational medium for dynamic control of growth and form. *Biosystems*, 164:76–93, 2017.

26 Nancy Lynch, Cameron Musco, and Merav Parter. Computational Tradeoffs in Biological Neural Networks: Self-Stabilizing Winner-Take-All Networks. In *Proceedings of the Conference on Innovations in Theoretical Computer Science (ITCS 2017)*, pages 15:1–15:44, 2017.

27 Wolfgang Maass. Networks of spiking neurons: the third generation of neural network models. *Neural networks*, 10(9):1659–1671, 1997.

28 Arik Motskin, Tim Roughgarden, Primoz Skraba, and Leonidas J. Guibas. Lightweight Coloring and Desynchronization for Networks. In *Proceedings of the of the Conference on Computer Communication (INFOCOM)*, 2009.

29 Saket Navlakha and Ziv Bar-Joseph. Algorithms in nature: the convergence of systems biology and computational thinking. *Molecular Systems Biology*, 7(1):546, 2011.

30 Saket Navlakha and Ziv Bar-Joseph. Distributed information processing in biological and computational systems. *Communications of the ACM*, 58(1):94–102, 2014.

31 Calvin Newport. Radio network lower bounds made easy. In *Proceedings of the International Symposium on Distributed Computing (DISC)*, pages 258–272. Springer, 2014.

32 Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

33 Alex Scott, Peter Jeavons, and Lei Xu. Feedback from nature: an optimal distributed algorithm for maximal independent set selection. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 147–156. ACM, 2013.

34 Alex Scott, Peter Jeavons, and Lei Xu. Feedback from nature: an optimal distributed algorithm for maximal independent set selection. In *Proceedings of the Symposium on Principles of Distributed Computing (PODC)*, 2013.

35 Asaf Shiloni, Noa Agmon, and Gal A Kaminka. Of robot ants and elephants: A computational comparison. *Theoretical Computer Science*, 412(41):5771–5788, 2011.

36 Hava T Siegelmann and Eduardo D Sontag. Turing computability with neural nets. *Applied Mathematics Letters*, 4(6):77–80, 1991.

37 David Soloveichik, Matthew Cook, Erik Winfree, and Jehoshua Bruck. Computation with finite stochastic chemical reaction networks. *natural computing*, 7(4):615–633, 2008.

Parallel Finger Search Structures

Seth Gilbert
Computer Science, National University of Singapore

Wei Quan Lim
Computer Science, National University of Singapore

Abstract

In this paper we present two versions of a **parallel finger structure** $\mathbb{FS}$ on p processors that supports searches, insertions and deletions, and has a finger at each end. This is to our knowledge the first implementation of a parallel search structure that is ***work-optimal*** with respect to the finger bound and yet has very good parallelism (within a factor of $O\big((\log p)^2\big)$ of optimal). We utilize an **extended implicit batching** framework that transparently facilitates the use of $\mathbb{FS}$ by any parallel program P that is modelled by a dynamically generated DAG D where each node is either a unit-time instruction or a call to $\mathbb{FS}$.

The work done by $\mathbb{FS}$ is bounded by the **finger bound** F_L (for some linearization L of D), i.e. each operation on an item with distance r from a finger takes $O(\log r + 1)$ amortized work. Running P using the simpler version takes $O\Big(\frac{T_1+F_L}{p} + T_\infty + d \cdot \big((\log p)^2 + \log n\big)\Big)$ time on a greedy scheduler, where T_1, T_∞ are the size and span of D respectively, and n is the maximum number of items in $\mathbb{FS}$, and d is the maximum number of calls to $\mathbb{FS}$ along any path in D. Using the faster version, this is reduced to $O\Big(\frac{T_1+F_L}{p} + T_\infty + d \cdot (\log p)^2 + s_L\Big)$ time, where s_L is the weighted span of D where each call to $\mathbb{FS}$ is weighted by its cost according to F_L. $\mathbb{FS}$ can be extended to a fixed number of movable fingers.

The data structures in our paper fit into the ***dynamic multithreading*** paradigm, and their performance bounds are directly ***composable*** with other data structures given in the same paradigm. Also, the results can be translated to practical implementations using work-stealing schedulers.

2012 ACM Subject Classification Theory of computation → Parallel algorithms; Theory of computation → Shared memory algorithms; Theory of computation → Parallel computing models

Keywords and phrases Parallel data structures, Multithreading, Dictionaries, Comparison-based Search, Distribution-sensitive algorithms

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.20

Related Version The full version of this paper is available at `https://arxiv.org/abs/1908.02741`.

Funding This research was supported in part by Singapore MOE AcRF Tier 1 grant T1 251RES1719.

Acknowledgements We would like to express our gratitude to our families and friends for their wholehearted support, to the kind reviewers who provided helpful feedback, and to all others who have given us valuable comments and advice.

1 Introduction

There has been much research on designing parallel programs and parallel data structures. The **dynamic multithreading paradigm** (see [12] chap. 27) is one common parallel programming model, in which algorithmic parallelism is expressed through parallel programming primitives such as fork/join (also spawn/sync), parallel loops and synchronized methods, but the program cannot stipulate any mapping from subcomputations to processors. This is the case with many parallel languages and libraries, such as Cilk dialects [18, 23], Intel TBB [28], Microsoft Task Parallel Library [30] and subsets of OpenMP [25].

Recently, Agrawal et al. [3] introduced the exciting ***modular design*** approach of **implicit batching**, in which the programmer writes a multithreaded parallel program that uses a ***black box*** data structure, treating calls to the data structure as basic operations, and also

© Seth Gilbert and Wei Quan Lim;
licensed under Creative Commons License CC-BY
33rd International Symposium on Distributed Computing (DISC 2019).
Editor: Jukka Suomela; Article No. 20; pp. 20:1–20:18

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

provides a data structure that supports batched operations. Given these, the runtime system automatically combines these two components together, buffering data structure operations generated by the program, and executing them in batches on the data structure.

This idea was extended in [4] to data structures that do not process only one batch at a time. In this **extended implicit batching framework**, the runtime system not only holds the data structure operations in a **parallel buffer**, to form the next **batch**, but also **notifies** the data structure on receiving the first operation in each batch. Independently, the data structure can at any point **flush** the parallel buffer to get the next batch.

This framework nicely supports ***pipelined*** batched data structures, since the data structure can decide when it is ready to get the next input batch from the parallel buffer. Furthermore, this framework makes it easy for us to build ***composable*** parallel algorithms and data structures with composable performance bounds. This is demonstrated by both the parallel working-set map in [4] and the parallel finger structure in this paper.

Finger Structures

The **map** (or **dictionary**) data structure, which supports inserts, deletes and searches/updates, collectively referred to as **accesses**, comes in many different kinds. A common implementation of a map is a balanced binary search tree such as an AVL tree or a red-black tree, which (in the comparison model) takes $O(\log n)$ worst-case cost per access for a tree with n items. There are also maps such as splay trees [29] that have amortized rather than worst-case performance bounds.

A **finger structure** is a special kind of map that comes with a **fixed finger** at each end and a (fixed) number of **movable fingers**, each of which has a key (possibly $-\infty$ or ∞ or between adjacent items in the map) that determines its position in the map, such that accessing items nearer the fingers is cheaper. For instance, the finger tree [20] was designed to have the finger property in the worst case; it takes $O(\log r + 1)$ steps per operation with finger distance r (Definition 1), so its total cost satisfies the finger bound (Definition 2).

▶ **Definition 1** (Finger Distance). *Define the* ***finger distance of accessing an item*** *x on a finger structure M to be the number of items from x to the nearest finger in M (including x), and the* ***finger distance of moving a finger*** *to be the distance moved.*

▶ **Definition 2** (Finger Bound). *Given any sequence L of N operations on a finger structure M, let F_L denote the* ***finger bound*** *for L, defined by $F_L = \sum_{i=1}^{N}(\log r_i + 1)$ where r_i is the finger distance of the i-th operation in L when L is performed on M.*

Main Results

We present, to the best of our knowledge, the first parallel finger structure. In particular, we design two parallel maps that are ***work-optimal*** with respect to the Finger Bound F_L (i.e. it takes $O(F_L)$ work) for some linearization L of the operations (that is consistent with the results), while having very good parallelism. (We assume that each key comparison takes $O(1)$ steps.) In this paper we focus on basic finger structures with just one fixed finger at each end (no movable fingers).

These parallel finger structures can be used by any parallel program P, whose actual execution is captured by a **program DAG** D, where each node is an instruction that finishes in $O(1)$ time or an access (insert/delete/search/update) to the finger structure M, called an M**-call**, that blocks until the result is returned, and each edge represents a dependency due to the parallel programming primitives.

The first design, called $\mathbb{FS}_1$, is a simpler one that processes accesses one batch at a time.

▶ **Theorem 3** ($\mathbb{FS}_1$ Performance)**.** *If P uses $\mathbb{FS}_1$ (as M), then its running time on p processes using any greedy scheduler (i.e. at each step, as many tasks are executed as are available, up to p) is $O\Big(\frac{T_1+F_L}{p}+T_\infty+d\cdot\big((\log p)^2+\log n\big)\Big)$ for some linearization L of M-calls in D, where T_1 is the number of nodes in D, and T_∞ is the number of nodes on the longest path in D, and d is the maximum number of M-calls on any path in D, and n is the maximum size of M.* [1]

Notice that if M is an ideal concurrent finger structure (i.e. one that takes $O(F_L)$ work), then running P using M on p processors according to the linearization L takes $\Omega(T_{opt})$ worst-case time where $T_{opt}=\frac{T_1+F_L}{p}+T_\infty$. Thus $\mathbb{FS}_1$ gives an essentially optimal time bound except for the "span term" $d\cdot\big((\log p)^2+\log n\big)$, which adds $O\big((\log p)^2+\log n\big)$ time per $\mathbb{FS}_1$-call along some path in D.

The second design, called $\mathbb{FS}_2$, uses a complex internal pipeline to reduce the "span term".

▶ **Theorem 4** ($\mathbb{FS}_2$ Performance)**.** *If P uses $\mathbb{FS}_2$, then its running time on p processes using any greedy scheduler is $O\Big(\frac{T_1+F_L}{p}+T_\infty+d\cdot(\log p)^2+s_L\Big)$ for some linearization L of M-calls in D, where T_1,T_∞,d are defined as in Theorem 3, and s_L is the weighted span of D where each $\mathbb{FS}_2$-call is weighted by its cost according to F_L. Specifically, each access operation on $\mathbb{FS}_2$ with finger distance r according to L is given the weight $\log r+1$, and s_L is the maximum weight of any path in D. Thus $\mathbb{FS}_2$ gives an essentially optimal time bound up to an extra $O\big((\log p)^2\big)$ time per $\mathbb{FS}_2$-call along some path in D.*

See the full paper for how to extend $\mathbb{FS}_1$ to a general finger structure with f movable fingers, and how to adapt the results for work-stealing schedulers.

Other Related Work

There are many approaches for designing efficient parallel data structures, to make maximal use of parallelism in a multi-processor system, whether with empirical or theoretical efficiency.

For example, Ellen et al. [15] show how to design a non-blocking concurrent binary search tree, with later work analyzing the amortized complexity [14] and generalizing this technique [11]. Another notable concurrent search tree is the CBTree [2, 1], which is based on the splay tree. But despite experimental success, the theoretical access cost for these tree structures may increase with the number of concurrent operations due to contention near the root, and some of them do not even maintain balance (i.e., the height may get large).

Another method is software combining [17, 21, 26], where each process inserts a request into a shared queue and at any time one process is sequentially executing the outstanding requests. This generalizes to parallel combining [6], where outstanding requests are executed in batches on a suitable batch-parallel data structure (similar to implicit batching). These methods were shown to yield empirically efficient concurrent implementations of various common abstract data structures including stacks, queues and priority queues.

In the PRAM model, Paul et al. [27] devised a parallel 2-3 tree where p synchronous processors can perform a sorted batch of p operations on a parallel 2-3 tree of size n in $O(\log n+\log p)$ time. Blelloch et al. [9] show how to increase parallelism of tree operations

[1] To cater to instructions that may not finish in $O(1)$ time (e.g. due to memory contention), it suffices to define T_1 and T_∞ to be the (weighted) work and span (Definition 5) respectively of the program DAG where each M-call is assumed to take $O(1)$ time.

via pipelining. Other similar data structures include parallel treaps [10] and a variety of work-optimal parallel ordered sets [7] supporting unions and intersections with optimal work, but these do not have optimal span. As it turns out, we can in fact have parallel ordered sets with optimal work and span [5, 24].

Nevertheless, the programmer cannot use this kind of parallel data structure as a black box in a high-level parallel program, but must instead carefully coordinate access to it. This difficulty can be eliminated by designing a suitable ***batch-parallel*** data structure and using ***implicit batching*** [3] or ***extended implicit batching*** as presented in [4]. Batch-parallel implementations have been designed for various data structures including weight-balanced B-trees [16], priority queues [6], working-set maps [4] and euler-tour trees [31].

2 Parallel Computation Model

In this section, we describe parallel programming primitives in our model, how a parallel program generates an execution DAG, and how we measure the cost of an execution DAG.

2.1 Parallel Primitives

The parallel finger structures $\mathbb{FS}_1$ and $\mathbb{FS}_2$ in this paper are described and explained as multithreaded data structures that can be used as composable building blocks in a larger parallel program. In this paper we shall focus on the abstract algorithms behind $\mathbb{FS}_1$ and $\mathbb{FS}_2$, relying merely on the following parallel programming primitives (rather than model-specific implementation details, but see the full paper for those):

1. **Threads**: A thread can at any point ***terminate*** itself (i.e. finish running). Or it can ***fork*** another thread, obtaining a pointer to that thread, or ***join*** to a previously forked thread (i.e. wait until that thread terminates). Or it can ***suspend*** itself (i.e. temporarily stop running), after which a thread with a pointer to it can ***resume*** it (i.e. make it continue running from where it left off). Each of these takes $O(1)$ time.
2. **Non-blocking locks**: Attempts to ***acquire*** a non-blocking lock are serialized but do not block. Acquiring the lock succeeds if the lock is not currently held but fails otherwise, and ***releasing*** always succeeds. If k threads concurrently access the lock, then each access finishes within $O(k)$ time.
3. **Dedicated lock**: A dedicated lock is a blocking lock initialized with a constant number of keys, where concurrent threads must use different keys to ***acquire*** it, but ***releasing*** does not require a key. Each attempt to acquire the lock takes $O(1)$ time, and the thread will acquire the lock after at most $O(1)$ subsequent acquisitions of that lock.
4. **Reactivation calls**: A procedure P with no input/output can be encapsulated by a reactivation wrapper, in which it can be run only via ***reactivations***. If there are always at most $O(1)$ concurrent reactivations of P, then whenever a thread ***reactivates*** P, if P is not currently running then it will start running (in another thread forked in $O(1)$ time), otherwise it will run within $O(1)$ time after its current run finishes.

We also make use of basic batch operations, namely filtering, sorted partitioning, joining and merging (see Appendix Appendix A.2), which have easy implementations using arrays in the CREW PRAM model. So $\mathbb{FS}_1$ and $\mathbb{FS}_2$ (using a work-stealing scheduler) can be implemented in the (synchronous) Arbitrary CRCW PRAM model with fetch-and-add, achieving the claimed performance bounds. Actually, $\mathbb{FS}_1$ and $\mathbb{FS}_2$ were also designed to function correctly with the same performance bounds in a much stricter computation model called the QRMW parallel pointer machine model (see Appendix Appendix A.1 for details).

2.2 Execution DAG

The **program DAG** D captures the high-level execution of P, but the actual complete execution of P (including interaction between data structure calls) is captured by the **execution DAG** E (which may be schedule-dependent), in which each node is a basic instruction and the directed edges represent the computation dependencies (such as constrained by forking/joining of threads and acquiring/releasing of blocking locks). At any point during the execution of P, a node in the program/execution DAG is said to be **ready** if its parent nodes have been executed. At any point in the execution, an **active thread** is simply a ready node in E, while a **terminated/suspended thread** is an executed node in E that has no child nodes.

The execution DAG E consists of **program nodes** (specifically P**-nodes**) and **ds (data-structure) nodes**, which are dynamically generated as follows. At the start E has a single program node, corresponding to the start of the program P. Each node could be a **normal instruction** (i.e. basic arithmetic/memory operation) or a **parallel primitive** (see Section 2.1). Each program node could also be a **data structure call**.

When a (ready) node is executed, it may generate child nodes or ***terminate***. A normal instruction generates one child node and no extra edges. A ***join*** generates a child node with an extra edge to it from the ***terminate*** node of the joined thread. A ***resume*** generates an extra child node (the resumed thread) with an edge to it from the ***suspend*** node of the originally suspended thread. Accesses to locks and reactivation calls would each expand to a subDAG comprised of normal instructions and possibly ***fork***/***suspend***/***resume***.

The program nodes correspond to nodes in the program DAG D, and except for data structure calls they generate only program nodes. A call to a data structure M is called an M**-call**. If M is an ordinary (non-batched) data structure, then an M-call generates an M**-node** (and every M-node is a ds node), which thereafter generates only M-nodes except for calls to other data structures (external to M) or returning the result of some operation (generating a program node with an edge to it from the original M-call).

However, if M is an ***(implicitly) batched*** data structure, then all M-calls are automatically passed to the **parallel buffer** for M (see Appendix Appendix A.3). So an M-call generates a **buffer node** corresponding to passing the call to the parallel buffer, as if the parallel buffer for M is itself another data structure and not part of M. Buffer nodes generate only buffer nodes until it notifies M of the buffered M-calls or passes the input batch to M, which generates an M-node. In short, M-nodes exclude all nodes generated as part of the buffer subcomputations (i.e. buffering the M-calls, and notifying M, and flushing the buffer).

2.3 Data Structure Costs

We shall now define work and span of any (terminating) subcomputation of a multithreaded program, i.e. any subset of the nodes in its execution DAG. This allows us to capture the intrinsic costs incurred by a data structure, separate from those of parallel programs using it.

▶ **Definition 5** (Subcomputation Work/Span/Cost). *Take any execution of a parallel program P (on p processors), and take any subset C of nodes in its execution DAG E. The **work** taken by C is the total weight w of C where each node is weighted by the time taken to execute it. The **span** taken by C is the maximum weight s of nodes in C on any (directed) path in E. The **cost** of C is $\frac{w}{p} + s$.*

▶ **Definition 6** (Data Structure Work/Span/Cost). *Take any parallel program P using a data structure M. The **work/span/cost** of M (as used by P) is the work/span/cost of the M-nodes in the execution DAG for P.*

Note that the cost of the entire execution DAG is in fact an upper bound on the actual time taken to run it on a **greedy scheduler**, which on each step assigns as many unassigned ready nodes (i.e. nodes that have been generated but have not been assigned) as possible to available processors (i.e. processors that are not executing any nodes) to be executed.

Moreover, the subcomputation cost is ***subadditive*** across subcomputations. Thus our results are ***composable*** with other algorithms and data structures in this model, since we actually show the following for some linearization L (where F_L, d, n, s_L are as defined in Section 1 Main Results, and N is the total number of calls to the parallel finger structure).

▶ **Theorem 7** ($\mathbb{FS}$ Work/Span Bounds).

- $\mathbb{FS}_1$ *takes* $O(F_L)$ *work and* $O\left(\frac{N}{p} + d \cdot \left((\log p)^2 + \log n\right)\right)$ *span.*
- $\mathbb{FS}_2$ *takes* $O(F_L)$ *work and* $O\left(\frac{N}{p} + d \cdot (\log p)^2 + s_L\right)$ *span.*

Note that the bounds for the work/span of $\mathbb{FS}_1$ and $\mathbb{FS}_2$ are independent of the scheduler. In addition, using any greedy scheduler, the parallel buffer for either finger structure has cost $O\left(\frac{T_1+F_L}{p} + d \cdot \log p\right)$ (Appendix Theorem 13). Therefore our main results (Theorem 3 and Theorem 4) follow from these composable bounds (Theorem 7).

3 Amortized Sequential Finger Structure

In this section we explain an amortized sequential finger structure $\mathbb{FS}_0$ with a fixed finger at each end, which is amenable to parallelization and pipelining due to its ***doubly-exponential segmented structure*** (which was partially inspired by Iacono's working-set structure [22]).

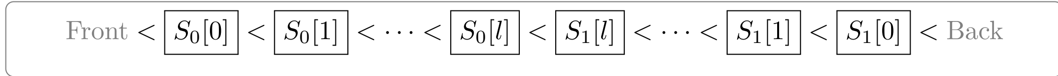


Figure 1 $\mathbb{FS}_0$ Outline; each box $S_i[k]$ represents a 2-3 tree of size $\Theta\left(2^{2^k}\right)$ for $k < l$.

$\mathbb{FS}_0$ keeps the items in order in two halves, the front half stored in a chain of **segments** $S_0[0..l]$, and the back half stored in reverse order in a chain of segments $S_1[0..l]$. Let $c(k) = 2^{2^{k+1}}$ for each $k \in \mathbb{Z}$. Each segment $S_i[k]$ has a **target size** $t(k) = 2 \cdot c(k)$, and a **target capacity** defined to be $[t(k), t(k)]$ if $k < l$ but $[0, t(k)]$ if $k = l$. Each segment stores its items in order in a 2-3 tree. We say that a segment $S_i[k]$ is **balanced** iff its size is within $c(k)$ of its target capacity, and **overfull** iff it has more than $c(k)$ items above target capacity, and **underfull** iff it has more than $c(k)$ items below target capacity. At any time we associate every item x to a unique segment that it **fits** in; x fits in $S_0[k]$ if k is the minimum such that $x \le \max(S_0[k])$, and that x fits in $S_1[k]$ if k is the minimum such that $x \ge \min(S_1[k])$, and that x fits in $S_0[l]$ if $\max(S_0[l]) < x < \min(S_1[l])$. We shall maintain the invariant that every segment is balanced after each operation is finished.

For each operation on an item x, we find the segment $S_i[k]$ that x fits in, by checking the range of items in $S_0[a]$ and $S_1[a]$ for each a from 0 to l and stopping once k is found, and then perform the desired operation on the 2-3 tree in $S_i[k]$. This takes $O(k + \log(t(k) + c(k))) \subseteq O(2^k) \subseteq O(\log r + 1)$ steps where r is the finger distance of the operation, since $\log_2 r + 1 \ge \log_2 c(k-1) = 2^k$.

After that, if $S_i[k]$ becomes imbalanced, we **rebalance** it by shifting (appropriate) items to or from $S_i[k+1]$ (after creating empty segment $S_i[k+1]$ if it does not exist) to make $S_i[k]$ have target size or as close as possible (via a suitable split then join of the 2-3 trees), and

then $S_i[k+1]$ is removed if it is the last segment and is now empty. After the rebalancing, $S_i[k]$ will not only be balanced but also have size within its target capacity. But now $S_i[k+1]$ may become imbalanced, so the rebalancing may cascade.

Finally, if one chain $S_i[0..l']$ is longer than the other chain $S_j[0..l]$, it must be that $l' = l+1$, so we **rebalance** the chains as follows: If $S_j[l]$ is below target size, shift items from $S_i[l']$ to $S_j[l]$ to fill it up to target size. If $S_j[l]$ is (still) below target size, remove the now empty $S_i[l']$, otherwise add a new empty segment $S_j[l+1]$.

Each rebalancing cascade may take $\Theta(\log n)$ steps, but the total rebalancing cost is only $O(1)$ amortized steps per operation, which we can prove via an accounting argument: We are given 1 credit for each operation, and use it to maintain a ***credit invariant*** that each segment $S_i[k]$ with q items beyond (i.e. above or below) its target capacity has at least $q \cdot 2^{-k}$ stored credits, and use the stored credits to pay for all rebalancing. Whenever a segment $S_i[k]$ is rebalanced, it must have had q items beyond its target capacity for some $q > c(k)$, and so had at least $q \cdot 2^{-k}$ stored credits. Also, the rebalancing itself takes $O(\log(t(k)+q) + \log(t(k+1)+c(k+1)+q)) \subseteq O(\log q) \subseteq O\big(q \cdot 2^{-k}\big)$ steps, after which $S_i[k+1]$ needs at most $q \cdot 2^{-(k+1)}$ extra stored credits. Thus the stored credits at $S_i[k]$ can be used to pay for both the rebalancing and any extra stored credits needed by $S_i[k+1]$. Whenever the chains are rebalanced, it can be paid for by the last segment rebalancing (which created or removed a segment), and no extra stored credits are needed.

4 Simpler Parallel Finger Structure

We now present our simpler parallel finger structure $\mathbb{FS}_1$. The idea is to use the amortized sequential finger structure $\mathbb{FS}_0$ (Section 3) and execute operations in batches. We group each pair of segments $S_0[k]$ and $S_1[k]$ into one **section** $S[k]$, and we say that an item x **fits in** the sections $S[j..k]$ iff x fits in some segment in $S[j..k]$.

Each segment is stored in an **optimal batch-parallel map** [24, 8], which supports:

- **Unsorted batch search:** Search for an unsorted batch of b items, tagging each search with the result, within $O(b \cdot \log n)$ work and $O(\log b \cdot \log n)$ span, where n is the map size.
- **Sorted batch access:** Perform an item-sorted batch of b operations on distinct items, tagging each operation with the result, within $O(b \cdot \log n)$ work and $O(\log b + \log n)$ span, where n is the map size before the batch access.
- **Split**: Split a map M of size k around a pivot rank r into maps M_1, M_2 where M_1 contains the first r items in M, and M_2 contains the last $k-r$ items in M, within $O(\log k)$ work/span.
- **Join**: Join maps M_1, M_2 of total size k where the greatest item in M_1 is less than the least item in M_2, within $O(\log k)$ work/span.

For each section $S[k]$, we can perform a batch of b operations on it within $O(b \cdot \log c(k))$ work and $O(\log b + \log c(k))$ span if we have the batch sorted. Excluding sorting, the total work would satisfy the finger bound just like in $\mathbb{FS}_0$. But we cannot afford to sort the input batch right at the start, because if the batch had b searches of distinct items all with finger distance $O(1)$, then it would take $\Omega(b \cdot \log b)$ work and exceed our finger bound budget of $O(b)$.

We can solve this by splitting the sections into two slabs, where the first slab comprises the first $\log \log(2b)$ sections, and passing the batch through a preliminary phase in which we merely perform an unsorted search of the relevant items in the first slab, and eliminate operations on items that fit in the first slab but are neither found nor to be inserted.

This preliminary phase takes $O(\log c(k))$ work per operation and $O(\log b \cdot \log c(k))$ span at each section $S[k]$. We then sort the uneliminated operations and execute them on the appropriate slab. For this, ordinary sorting still takes too much work as there can be many

operations on the same item, but it turns out that the finger bound budget is enough to pay for entropy-sorting (Appendix Definition 15), which takes $O\left(\log \frac{b}{q} + 1\right)$ work for each item that occurs q times in the batch. Rebalancing the segments and chains is a little tricky, but if done correctly it takes $O(1)$ amortized work per operation. Therefore we achieve work-optimality while being able to process each batch within $O\left((\log b)^2 + \log n\right)$ span. The details are below.

4.1 Description of $\mathbb{FS}_1$

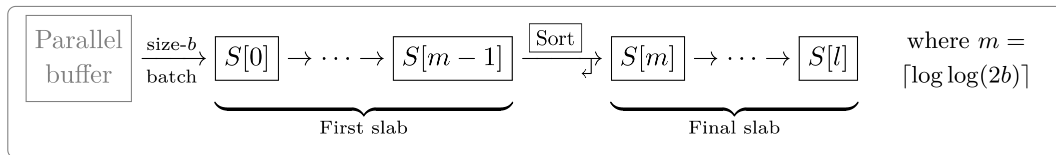


Figure 2 $\mathbb{FS}_1$ Outline; each batch is sorted only after being filtered through the smaller sections.

$\mathbb{FS}_1$-calls are put into the parallel buffer (Section 2) for $\mathbb{FS}_1$. Whenever the previous batch is done, $\mathbb{FS}_1$ flushes the parallel buffer to obtain the next batch B. Let b be the size of B, and we can assume $b > 1$. Based on b, the sections in $\mathbb{FS}_1$ are conceptually divided into two slabs, the **first slab** comprising sections $S[0..m-1]$ and the **final slab** comprising sections $S[m..l]$, where $m = \lceil \log \log(2b) \rceil + 1$ (where log is the binary logarithm). The items in each segment are stored in a batch-parallel map.

$\mathbb{FS}_1$ processes the input batch B in four phases:

1. **Preliminary phase**: For each first slab section $S[k]$ in order (i.e. k from 0 to $m-1$) do as follows:
   a. Perform an unsorted search in each segment in $S[k]$ for all the items relevant to the remaining batch B' (of direct pointers into B), and tag the operations in the original batch B with the results.
   b. Remove all operations on items that fit in $S[k]$ from the remaining batch B'.
   c. Skip the rest of the first slab if B' becomes empty.
2. **Separation phase**: Partition B based on the tags into three parts and handle each part separately as follows:
   a. ***Ineffectual operations*** (on items that fit in the first slab but are neither found nor to be inserted): Return the results.
   b. ***Effectual operations*** (on items found in or to be inserted into the first slab): Entropy-sort (Appendix Definition 15) them in order of access type (search, update, insertion, deletion) with deletions last, followed by item, combining operations of the same access type on the same item into one **group-operation** that is treated as a single operation whose **effect** is the last operation in that group. Each group-operation is stored in a leaf-based binary tree with height $O(\log b)$ (but not necessarily balanced), and the combining is done during the entropy-sorting itself.
   c. ***Residual operations*** (on items that do not fit in the first slab): Sort them while combining operations in the same manner as for effectual operations.

3. **Execution phase**: Execute the effectual operations as a batch on the first slab, and then execute the residual operations as a batch on the final slab, for each slab doing the following at each section $S[k]$ in order (small to big):
   a. Let $G_{1..4}$ be the partition of the batch of operations into the 4 access types (deletions last), each G_a sorted by item.
   b. For each segment $S_i[k]$ in $S[k]$, and for each a from 1 to 4, cut out the operations that fit in $S_i[k]$ from G_a, and perform those operations (as a sorted batch) on $S_i[k]$, and then return their results.
   c. Skip the rest of the slab if the batch becomes empty.
4. **Rebalancing phase**: Rebalance all the segments and chains, by doing the following:
   a. ***Segment rebalancing***: For each chain S_i, for each segment $S_i[k]$ in S_i in order (small to big):
      i. If $k > 0$ and $S_i[k-1]$ is overfull, make $S_i[k-1]$ have target size by shifting items from it to $S_i[k]$.
      ii. If $k > 0$ and $S_i[k-1]$ is underfull and $S_i[k]$ has at least $\frac{c(k)}{2}$ items, let $S_i[k']$ be the first underfull segment in S_i, and **fill** $S_i[k'..k-1]$ using $S_i[k]$ as follows: for each j from $k-1$ down to k', shift items from $S_i[j+1]$ to $S_i[j]$ to make $S_i[k'..j]$ have total size $\sum_{a=k'}^{j} t(a)$ or as close as possible, and then remove $S_i[j+1]$ if it is emptied.
      iii. If $S_i[k]$ is (still) overfull and is the last segment in S_i, create a new (empty) segment $S_i[k+1]$.
      iv. Skip the rest of the current slab if $S_i[k]$ is balanced and the execution phase had skipped $S[k]$.
   b. ***Chain rebalancing***: After that, if one chain S_i is longer than the other chain S_j, repeat the following until the chains are the same length:
      i. Let the current chains be $S_i[0..k]$ and $S_j[0..k']$. Create new (empty) segments $S_j[k'+1..k]$, and shift all items from $S_i[k]$ to $S_j[k]$, and then **fill** the underfull segments in $S_j[k'..k-1]$ using $S_j[k]$ (as in step 4aii). If $S_j[k]$ is (now) empty again, remove $S[k]$.

4.2 Analysis of $\mathbb{FS}_1$

It is not hard to prove that every segment is balanced (just) after the rebalancing phase. (See the full paper for the details.) Based on that, we shall now bound the work done by $\mathbb{FS}_1$.

▶ **Definition 8** (Inward Order). *Take any sequence A of map operations and let I be the set of items accessed by operations in A. Define the **inward distance** of an operation in A on an item x to be* $\min(\text{size}(I_{\leq x}), \text{size}(I_{\geq x}))$. *We say that A is in **inward order** iff its operations are in order of (non-strict) increasing inward distance. Naturally, we say that A is in **outward order** iff its reverse is in inward order.*

▶ **Theorem 9** ($\mathbb{FS}_1$ Work). *$\mathbb{FS}_1$ takes $O(F_L)$ work for some linearization L of $\mathbb{FS}_1$-calls in D.*

Proof. Let L^* be a linearization of $\mathbb{FS}_1$-calls in D such that:
- Operations on $\mathbb{FS}_1$ in earlier input batches are before those in later input batches.
- The operations within each batch are ordered as follows:
  1. Ineffectual operations are before effectual/residual operations.
  2. Effectual/residual operations are in order of access type (deletions last).
  3. Effectual insertions are in inward order, and effectual deletions are in outward order.
  4. Operations in each group-operation are consecutive and in the same order as in that group.

Let L' be the same as L^* except that in point 3 effectual deletions are ordered so that those on items in earlier sections are later (instead of outward order). Now consider each input batch B of b operations on $\mathbb{FS}_1$.

In the preliminary and execution phases, each section $S[a]$ takes $O(2^a)$ work per operation. Thus each operation in B with finger distance r according to L' on an item x that was found to fit in section $S[k]$ takes $O\left(\sum_{a=0}^{k} 2^a\right) = O(2^k) \subseteq O(\log r + 1)$ work, because $r \geq \sum_{a=0}^{k-1} c(a) + 1 \geq \frac{1}{2}c(k-1)$ if $S[k]$ is in the first slab (since earlier effectual operations in B did not delete items in $S[0..k-1]$), and $r \geq \sum_{a=0}^{k-1} c(a) - b \geq \frac{1}{2}c(k-1)$ if $S[k]$ is in the final slab (since $b \leq \frac{1}{2}c(m-1)$). Therefore these phases take $O(F_{L'})$ work in total.

Let G be the effectual operations in B as a subsequence of L^*. Entropy-sorting G takes $O(H + b)$ work (Appendix Theorem 16), where H is the entropy of G (i.e. $H = \sum_{i=1}^{b} \log \frac{b}{q_i}$ where q_i is the number of occurrences of the i-th operation in G). Partition G into 3 parts: searches/updates G_1 and insertions G_2 and deletions G_3. And let H_j be the entropy of G_j. Then $H = \sum_{j=1}^{3} H_j + \sum_{i=1}^{b} \log \frac{b}{b_i}$ where b_i is the number of operations in the same part of G as the i-th operation in G, and $\sum_{i=1}^{b} \log \frac{b}{b_i} \leq b \cdot \log\left(\frac{1}{b} \sum_{i=1}^{b} \frac{b}{b_i}\right) = b \cdot \log 3$ by Jensen's inequality. Thus entropy-sorting G takes $O\left(\sum_{j=1}^{3} H_j + b\right)$ work. Let C_j be the cost of G_j according to F_{L^*}. Since each operation in G_j has inward distance (with respect to G_j) at most its finger distance according to L^*, we have $H_j \in O(C_j)$ (Appendix Theorem 14), and hence entropy-sorting takes $O(F_{L^*})$ work in total.

Sorting the residual operations in B (that do not fit in the first slab) takes $O(\log b) \subseteq O(\log r)$ work per operation with finger distance r according to L^*, since $r \geq c(m-1) \geq 2b$.

Therefore the separation phase takes $O(F_{L^*})$ work in total. Finally, the rebalancing phase takes $O(1)$ amortized work per operation, as we shall prove in the next lemma. Thus $\mathbb{FS}_1$ takes $O(\max(F_{L^*}, F_{L'}))$ total work. ◀

▶ **Lemma 10** ($\mathbb{FS}_1$ Rebalancing Work)**.** *The rebalancing phase of $\mathbb{FS}_1$ takes $O(1)$ amortized work per operation.*

Proof. We shall maintain the credit invariant that each segment $S_i[k]$ with q items beyond its target capacity has at least $q \cdot 2^{-k}$ stored credits. The execution phase clearly increases the total stored credits needed by at most 1 per operation, which we can pay for. We now show that the invariant can be preserved after the segment rebalancing and the chain rebalancing.

During the segment rebalancing (step 4a), each shift is performed between some neighbouring segments $S_i[k]$ and $S_i[k+1]$, where $S_i[k]$ has $t(k)+q$ items and $S_i[k+1]$ has $t(k+1)+q'$ items just before the shift, and $|q| > c(k)$. The shift clearly takes $O(\log(t(k)+q)+\log(t(k+1)+q'))$ work. If $q' < 2 \cdot t(k+1)$ then this is obviously just $O(\log t(k) + \log |q|)$ work. But if $q' > 2 \cdot t(k+1)$, then $S_i[k+1]$ will also be rebalanced in step 4ai of the next segment balancing iteration, since at most $\sum_{a=0}^{k} t(a) \leq t(k+1)$ items will be shifted from $S_i[k+1]$ to $S_i[k]$ in step 4aii, and hence $S_i[k+1]$ will still have at least q' items. In that case, the second term $O(\log(t(k+1) + q'))$ in the work bound for this shift can be bounded by the first term of the work bound for the subsequent shift from $S_i[k+1]$ to $S_i[k+2]$, since $\log(t(k+1) + q') \in O(\log q')$. Therefore in any case we can treat this shift as taking only $O(\log t(k) + \log |q|) \subseteq O(\log |q|) \subseteq O\left(|q| \cdot 2^{-k}\right)$ work.

Now consider the two kinds of segment rebalancing:

- *Overflow*: step 4ai shifts items from overfull $S_i[k]$ to $S_i[k+1]$, where $S_i[k]$ has $t(k) + u$ items just before the shift. After the shift, $S_i[k]$ has target size and needs no stored credits, and $S_i[k+1]$ would need at most $u \cdot 2^{-(k+1)}$ extra stored credits. Thus the $u \cdot 2^{-k}$ credits stored at $S_i[k]$ can pay for both the shift and the needed extra stored credits.

- *Fill*: step 4aii fills some underfull segments $S_i[k'..k]$ using $S_i[k+1]$, where $S_i[j]$ has $t(j) - u_i(j)$ items just before the fill, for each $j \in [k'..k]$. After the fill, every segment in $S_i[k'..k]$ would have target size and need no stored credits, and $S_i[k+1]$ will need at most $\left(\sum_{j=k'}^{k} u_i(j)\right) \cdot 2^{-(k+1)} \leq \frac{1}{2}\sum_{j=k'}^{k}\left(u_i(j) \cdot 2^{-j}\right)$ extra stored credits, which can be paid for by using half the credits stored at each segment in $S_i[k'..k]$. The other half of the $u_i(j) \cdot 2^{-j}$ credits stored at $S_i[j]$ suffices to pay for the shift from $S_i[j+1]$ to $S_i[j]$ for each $j \in [k'..k]$.

If chain rebalancing (step 4b) is performed, segment rebalancing must have created or removed some segment, in which case there were enough deletions or shifted items that the work done by chain rebalancing can be ignored. The details are in the full paper. ◀

The span bound for $\mathbb{FS}_1$ is also relegated to the full paper.

5 Faster Parallel Finger Structure

Although $\mathbb{FS}_1$ has optimal work and a small span, it is possible to reduce the span even further, intuitively by pipelining the batches in some fashion so that an expensive access in a batch does not hold up the next batch.

As with $\mathbb{FS}_1$, we need to split the sections into two slabs, but this time we fix the first slab at m sections where $m \in \log \Theta(\log p)$ so that we can pipeline just the final slab. We need to allow big enough batches so that operations that are delayed because earlier batches are full can count their delay against the total work divided by p. But to keep the span of the sorting phase down to $O\big((\log p)^2\big)$, we need to restrict the batch size. It turns out that restricting to batches of size at most p^2 works.

We cannot pipeline the first slab (particularly the rebalancing), but the preliminary phase and separation phase would only take $O\big((\log p)^2\big)$ span. The execution phase and rebalancing phases are still carried out as before on the first slab, taking $O\big((\log p)^2\big)$ span, but execution and rebalancing on the final slab are pipelined, by having each final slab section $S[k]$ process the batch passed to it and rebalance the preceding segments $S_0[k-1]$ and $S_1[k-1]$ if necessary.

One ***key challenge*** is how to guarantee that such local rebalancing in the final slab is always possible and always sufficient. To ensure that, we do not allow $S[k]$ to proceed if it is imbalanced or if there are more than $c(k)$ pending operations in the buffer to $S[k+1]$. In such a situation, $S[k]$ must stop and reactivate $S[k+1]$, which would clear its buffer and rebalance $S[k]$ before restarting $S[k]$. It may be that $S[k+1]$ also cannot proceed for the same reason and is stopped in the same manner, and so $S[k]$ may be delayed by such a stop for a long time. But by a suitable accounting argument we can bound the total delay due to all such stops by the total work divided by p. Similarly, we do not allow the first slab to run (on a new batch) if $S[m-1]$ is imbalanced or there are more than $c(m-1)$ pending operations in the buffer to $S[m]$.

Finally, we use an odd-even locking scheme to ensure that the segments in the final slab do not interfere with each other yet can proceed at a consistent pace. The details are below.

5.1 Description of $\mathbb{FS}_2$

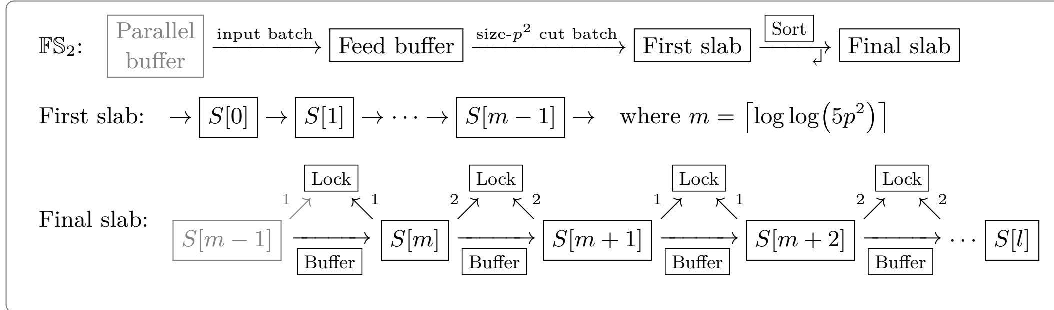


Figure 3 $\mathbb{FS}_2$ Sketch; the final slab is pipelined, facilitated by locks between adjacent sections.

We will need the **bunch** structure (Appendix Definition 12) for aggregating batches, which is an unsorted set supporting both addition of a batch of new elements within $O(1)$ work/span and conversion to a batch within $O(b)$ work and $O(\log b)$ span if it has size b.

$\mathbb{FS}_2$ has the same sections as in $\mathbb{FS}_1$, with the **first slab** comprising the first $m = \lceil \log\log(5p^2) \rceil$ sections, and the **final slab** comprising the other sections. $\mathbb{FS}_2$ uses a **feed buffer**, which is a queue of bunches each of size p^2 except the last (which can be empty). Whenever $\mathbb{FS}_2$ is notified of input (by the parallel buffer), it reactivates the first slab.

Each section $S[k]$ in the final slab has a **buffer** before it (for pending operations from $S[k-1]$), which for each access type uses an optimal batch-parallel map to store bunches of group-operations of that type, where operations on the same item are in the same bunch. When a batch of group-operations on an item is inserted into the buffer, it is simply added to the correct bunch. Whenever we count operations in the buffer, we shall count them individually even if they are on the same item. The first slab and each final slab section also has a **deferred flag**, which indicates whether its run is deferred until the next section has run. Between every pair of consecutive sections starting from after $S[m-1]$ is a **neighbour-lock**, which is a dedicated lock (see Section 2.1) with 1 key for each arrow to it in Figure 3.

Whenever the first slab is reactivated, it runs as follows:

1. If the parallel buffer and feed buffer are both empty, terminate.
2. Acquire the neighbour-lock between $S[m-1]$ and $S[m]$. (Skip steps 2 to 4 and steps 8 to 10 if $S[m]$ does not exist.)
3. If $S[m-1]$ has any imbalanced segment or $S[m]$ has more than $c(m-1)$ operations in its buffer, set the first slab's deferred flag and release the neighbour-lock, and then reactivate $S[m]$ and terminate.
4. Release the neighbour-lock.
5. Let q be the size of the last bunch F in the feed buffer. Flush the parallel buffer (if it is non-empty) and cut the input batch of size b into small batches of size p^2 except possibly the first and last, where the first has size $\min(b, p^2 - q)$. Add that first small batch to F, and append the rest as bunches to the feed buffer.
6. Remove the first bunch from the feed buffer and convert it into a batch B, which we call a **cut batch**.

7. Process B using the same four phases as in $\mathbb{FS}_1$ (Section 4.1), but restricted to the first slab (i.e. execute only the effectual operations on the first slab, and do segment rebalancing only on the first slab, and do chain rebalancing only if $S[m]$ had not existed before this processing). Furthermore, do not update $S[m-1]$'s segments' sizes until after this processing (so that section $S[m]$ in step 4 will not find any of $S[m-1]$'s segments imbalanced until the first slab rebalancing phase has finished).
8. Acquire the neighbour-lock between $S[m-1]$ and $S[m]$.
9. Insert the residual group-operations (on items that do not fit in the first slab) into the buffer of $S[m]$, and then reactivate $S[m]$.
10. Release the neighbour-lock.
11. Reactivate itself.

Whenever a final slab section $S[k]$ is reactivated, it runs as follows:

1. Acquire the neighbour-locks (between $S[k]$ and its neighbours) in the order given by the arrow number in Figure 3.
2. If $S[k]$ has any imbalanced segment or $S[k+1]$ (exists and) has more than $c(k)$ operations in its buffer, set $S[k]$'s deferred flag and release the neighbour-locks, and then reactivate $S[k+1]$ and terminate.
3. For each access type, flush and process the (sorted) batch G of bunches of group-operations of that type in its buffer as follows:
   a. Convert each bunch in G to a batch of group-operations.
   b. For each segment $S_i[k]$ in $S[k]$, cut out the group-operations on items that fit in $S_i[k]$ from G, and perform them (as a sorted batch) on $S_i[k]$, and then fork to return the results of the operations (according to the order within each group-operation).
   c. If G is non-empty (i.e. has leftover group-operations), insert G into the buffer of $S[k+1]$ and then reactivate $S[k+1]$.
4. Rebalance locally as follows (essentially like in $\mathbb{FS}_1$):
   a. For each segment $S_i[k]$ in $S[k]$:
      i. If $S_i[k-1]$ is overfull, shift items from $S_i[k-1]$ to $S_i[k]$ to make $S_i[k-1]$ have target size.
      ii. If $S_i[k-1]$ is underfull, shift items from $S_i[k]$ to $S_i[k-1]$ to make $S_i[k-1]$ have target size, and then remove $S_i[k]$ if it is emptied.
      iii. If $S_i[k]$ is (still) overfull and is the last segment in S_i, create a new segment $S_i[k+1]$ and reactivate it.
   b. If $S[k]$ is (still) the last section, but chain S_i is longer than chain S_j:
      i. Create a new segment $S_j[k]$ and shift all items from $S_i[k]$ to $S_j[k]$.
      ii. If $S_j[k-1]$ is (now) underfull, shift items from $S_j[k]$ to $S_j[k-1]$ to make $S_j[k-1]$ have target size.
      iii. If $S_j[k]$ is (now) empty again, remove $S[k]$.
5. If $k=m$, and the first slab is deferred, clear its deferred flag then reactivate it.
6. If $k>m$, and $S[k-1]$ is deferred, clear its defered flag then reactivate it.
7. Release the neighbour-locks.

5.2 Analysis of $\mathbb{FS}_2$

See the full paper for the proofs. The first step is to establish the $\mathbb{FS}_2$ Balance Invariants:

▶ **Lemma 11** ($\mathbb{FS}_2$ Balance Invariants)**.** *$\mathbb{FS}_2$ satisfies the following invariants:*

1. *When the first slab is not running, every segment in $S_i[0..m-2]$ is balanced and $S_i[m-1]$ has at most $2 \cdot t(m-1)$ items.*
2. *When a final slab section $S[k]$ rebalances a segment in $S[k-1]$ (in step 4a), it will make that segment have size $t(k-1)$.*
3. *Just after the last section finishes running without creating new sections, the segments in $S[k]$ are balanced and both chains have the same length.*
4. *Each final slab section $S[k]$ always has at most $2 \cdot c(k-1)$ operations in its buffer.*
5. *Each final slab segment $S_i[k]$ always has at most $2 \cdot t(k)$ items, and at least $c(k-1)$ items unless $S[k]$ is the last section.*

Using these invariants, we can prove the work bound for $\mathbb{FS}_2$ (as stated in Theorem 7), by linearizing operations that finish during the first slab run or final slab section according to when that run finishes, and linearizing operations that finish in the same first slab run in the same way as in the proof for $\mathbb{FS}_1$ (see Theorem 9). This relies on a supporting lemma that all rebalancing done by $\mathbb{FS}_2$ takes $O(1)$ amortized work per operation, which can be proven by a credit invariant like the one used for $\mathbb{FS}_1$ (see Lemma 10): Each segment $S_i[k]$ with q items beyond its target capacity has at least $q \cdot 2^{-k}$ stored credits, and each unfinished operation carries 1 credit with it.

The span bound for $\mathbb{FS}_2$ (as stated in Theorem 7) requires another credit invariant: For $k \geq m-1$, each segment $S_i[k]$ with q items beyond its target capacity has at least $q \cdot 2^{-k}$ stored credits, and each operation in $S[k+1]$'s buffer carries 2^{-k} credits with it. This invariant is used to bound the deferment delay (the delay that an operation may face due to deferred section runs), where we use the credits to pay for p times the deferment delay, to show that the deferment delay is at most $O\left(\frac{1}{p}\right)$ per operation on $\mathbb{FS}_2$. The rest of the delay incurred by each operation can be bounded by tracing its path through the sections and using the fact that the neighbour-locking scheme ensures that the first slab contributes $O\big((\log p)^2\big)$ delay and each final slab section $S[k]$ contributes $O\big(2^k\big)$ delay.

References

1 Yehuda Afek, Haim Kaplan, Boris Korenfeld, Adam Morrison, and Robert E Tarjan. The CB tree: a practical concurrent self-adjusting search tree. *Distributed computing*, 27(6):393–417, 2014.

2 Yehuda Afek, Haim Kaplan, Boris Korenfeld, Adam Morrison, and Robert Endre Tarjan. CBTree: A Practical Concurrent Self-Adjusting Search Tree. In *DISC*, volume 7611 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2012.

3 Kunal Agrawal, Jeremy T Fineman, Kefu Lu, Brendan Sheridan, Jim Sukha, and Robert Utterback. Provably good scheduling for parallel programs that use data structures through implicit batching. In *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*, pages 84–95. ACM, 2014.

4 Kunal Agrawal, Seth Gilbert, and Wei Quan Lim. Parallel Working-Set Search Structures. In *Proceedings of the 30th ACM symposium on Parallelism in algorithms and architectures*, pages 321–332. ACM, 2018. arXiv:1805.05787.

5 Yaroslav Akhremtsev and Peter Sanders. Fast parallel operations on search trees. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*, pages 291–300. IEEE, 2016.

6. Vitaly Aksenov, Petr Kuznetsov, and Anatoly Shalyto. Parallel Combining: Benefits of Explicit Synchronization. In Jiannong Cao, Faith Ellen, Luis Rodrigues, and Bernardo Ferreira, editors, *22nd International Conference on Principles of Distributed Systems (OPODIS 2018)*, volume 125 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:16, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.OPODIS.2018.11`.
7. Guy E Blelloch, Daniel Ferizovic, and Yihan Sun. Just join for parallel ordered sets. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 253–264. ACM, 2016.
8. Guy E Blelloch, Jeremy T Fineman, Yan Gu, and Yihan Sun. Optimal Parallel Algorithms in the Binary-Forking Model. *arXiv preprint*, 2019. `arXiv:1903.04650`.
9. Guy E. Blelloch and Margaret Reid-Miller. Pipelining with Futures. In *Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, SPAA '97, pages 249–259, New York, NY, USA, 1997. ACM. `doi:10.1145/258492.258517`.
10. Guy E. Blelloch and Margaret Reid-Miller. Fast Set Operations Using Treaps. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 16–26, 1998. `doi:10.1145/277651.277660`.
11. Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. In *ACM SIGPLAN Notices*, volume 49, pages 329–342. ACM, 2014.
12. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* The MIT Press, third edition, 2009.
13. Cynthia Dwork, Maurice Herlihy, and Orli Waarts. Contention in shared memory algorithms. *Journal of the ACM (JACM)*, 44(6):779–805, 1997.
14. Faith Ellen, Panagiota Fatourou, Joanna Helga, and Eric Ruppert. The amortized complexity of non-blocking binary search trees. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, pages 332–340. ACM, 2014.
15. Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking Binary Search Trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, pages 131–140, New York, NY, USA, 2010. ACM. `doi:10.1145/1835698.1835736`.
16. Stephan Erb, Moritz Kobitzsch, and Peter Sanders. Parallel bi-objective shortest paths using weight-balanced b-trees with bulk updates. In *International Symposium on Experimental Algorithms*, pages 111–122. Springer, 2014.
17. Panagiota Fatourou and Nikolaos D. Kallimanis. Revisiting the combining synchronization technique. In *PPoPP*, pages 257–266, 2012. `doi:10.1145/2145816.2145849`.
18. Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 212–223, 1998.
19. Michael T Goodrich and S Rao Kosaraju. Sorting on a parallel pointer machine with applications to set expression evaluation. *Journal of the ACM (JACM)*, 43(2):331–361, 1996.
20. Leo J Guibas, Edward M McCreight, Michael F Plass, and Janet R Roberts. A new representation for linear lists. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 49–60. ACM, 1977.
21. Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 355–364, 2010. `doi:10.1145/1810479.1810540`.
22. John Iacono. Alternatives to splay trees with O(log n) worst-case access times. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 516–522. Society for Industrial and Applied Mathematics, 2001.

23 Intel Corporation. *Intel Cilk Plus Language Extension Specification, Version 1.1*, 2013. Document 324396-002US. Available from `http://cilkplus.org/sites/default/files/open_specifications/Intel_Cilk_plus_lang_spec_2.htm`.

24 Wei Quan Lim. Optimal Multithreaded Batch-Parallel 2-3 Trees. *arXiv*, 2019. `arXiv:1905.05254`.

25 OpenMP Architecture Review Board. OpenMP application program interface, version 4.0. Available from `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`, July 2013.

26 Y. Oyama, K. Taura, and A. Yonezawa. Executing Parallel Programs With Synchronization Bottlenecks Efficiently. In *Proceedings of the International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications (PDSIA)*, pages 182–204, 1999.

27 Wolfgang Paul, Uzi Vishkin, and Hubert Wagener. Parallel dictionaries on 2–3 trees. *Automata, Languages and Programming*, pages 597–609, 1983.

28 James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly, 2007.

29 Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM (JACM)*, 32(3):652–686, 1985.

30 The Task Parallel Library. `http://msdn.microsoft.com/en-us/magazine/cc163340.aspx`, October 2007. URL: `http://msdn.microsoft.com/en-us/magazine/cc163340.aspx`.

31 Thomas Tseng, Laxman Dhulipala, and Guy Blelloch. Batch-Parallel Euler Tour Trees. In *2019 Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 92–106. SIAM, 2019.

A Appendix

Here we spell out the model details, building blocks and supporting theorems used in our paper. More details and proofs can be found in the full paper.

A.1 QRMW Pointer Machine Model

QRMW stands for **queued read-modify-write**, as described in [13]. In this contention model, asynchronous processors perform memory accesses via read-modify-write (RMW) operations (including read, write, test-and-set, fetch-and-add, compare-and-swap), which are supported by almost all modern architectures. Also, to capture contention costs, multiple memory requests to the same memory cell are FIFO-queued and serviced one at a time, and the processor making each memory request is blocked until the request has been serviced.

The QRMW pointer machine model, introduced in [4], extends the parallel pointer machine model in [19] to RMW operations. An RMW operation can be performed on any memory cell via a pointer to the memory node that it belongs to. All operations except for memory accesses take one step each. Accesses to each memory cell are FIFO-queued to be serviced, and the first access in the queue (if any) is serviced at each time step. The processor making each memory request is blocked until the request has been serviced.

A.2 Parallel Batch Operations

We rely on the following basic operations on batches:

- Split a given batch of n items into left and right parts around a given position, within $O(\log n)$ work/span.
- Partition a given batch of n items into lower and upper parts around a given pivot, within $O(n)$ work and $O(\log n)$ span.
- Partition a sorted batch of n items around a sorted batch of k pivots, within $O(k \cdot \log n)$ work and $O(\log n + \log k)$ span.

- Join a batch of batches with n total items, within $O(n)$ work and $O(\log n)$ span.
- Merge two sorted batches with n total items, optionally combining duplicates, within $O(n)$ work and $O(\log n)$ span if the combining procedure takes $O(1)$ work/span.

These can be implemented in the QRMW pointer machine model [24] with each batch stored as a **BBT** (leaf-based balanced binary tree). They can also be implemented (more easily) in the binary forking model in [8] with each batch stored in an array.

We also rely on the bunch data structure, defined as follows.

▶ **Definition 12** (Bunch Structure). *A **bunch** is an unsorted set supporting addition of any batch of new elements within $O(1)$ work/span and conversion to a batch within $O(b)$ work and $O(\log b)$ span if it has size b. A bunch can be implemented using a complete binary tree with batches at the leaves, with a linked list threaded through each level to support adding a new batch as a leaf in $O(1)$ work/span. To convert a bunch to a batch, we treat the bunch as a batch of batches and parallel join all the batches.*

A.3 Parallel Buffer

To facilitate extended implicit batching, we can use any parallel buffer implementation that takes $O(p + b)$ work and $O(\log p + \log b)$ span per batch of size b (on p processors), as long as any operation that arrives is (regardless of the scheduler) within $O(1)$ span included in the batch that is being flushed or in the next batch, and there are always at most $\frac{1}{2}p + q$ ready buffer nodes (active threads of the buffer) where q is the number of operations that are currently buffered or being flushed. This would entail the following parallel buffer overhead [4].

▶ **Theorem 13** (Parallel Buffer Cost). *Take any program P using an implicitly batched data structure M, run using any greedy scheduler. Then the cost (Definition 6) of the parallel buffer for M is $O\left(\frac{T_1+w}{p} + d \cdot \log p\right)$, where T_1 is the work of the P-nodes, and w is the work taken by M, and d is the maximum number of M-calls on any path in the program DAG D.*

▶ Remark. In general, if a program uses a fixed number of implicitly batched data structures, then running it using a greedy scheduler takes $O\left(\frac{T_1+w^*}{p} + T_\infty + s^* + d^* \cdot \log p\right)$ time, where w^* is the total work of all the data structures, and s^* is the total span of all the data structures, and d^* is the maximum number of data structure calls on any path in the program DAG.

The parallel buffer for each data structure M can be implemented using a static BBT, with a sub-buffer at each leaf node, one per processor, and a flag at each internal node. Each sub-buffer stores its operations as the leaves of a complete binary tree with a linked list through each level. Whenever a thread τ makes a call to M, the processor running τ suspends it and inserts the call together with a callback (i.e. a structure with a pointer to τ and a field for the result) into the sub-buffer for that processor. Then the processor walks up the BBT from leaf to root, test-and-setting each flag along the way, terminating if it was already set. On reaching the root, the processor notifies M (by reactivating it). M can eventually return the result of the call via the callback (i.e. by updating the result field and then resuming τ).

Whenever the buffer is flushed (by M), all sub-buffers are swapped out by a parallel recursion on the BBT, replaced by new sub-buffers in a newly constructed static BBT. We then wait for all pending insertions into the old sub-buffers to be completed, before joining their contents into a single batch to be returned (to M). To do so, each processor i has a flag y_i initialized to *true*, and a thread field ϕ_i initialized to *null*. Whenever it inserts an

M-call X, it sets $y_i := false$, then inserts X into the (current) sub-buffer, then resumes ϕ_i if TestAndSet$(y_i) = true$. To wait for pending insertions into the old sub-buffer for processor i, we store a pointer to the current thread in ϕ_i and then suspend it if TestAndSet$(y_i) = false$.

A.4 Sorting Theorems

Let S be the set of possible items, linearly ordered by a given comparison function.

▶ **Theorem 14** (Maximum Finger Bound). *Take any sequence I in S^n with in-order item frequencies $q_{1..u}$, namely the i-th smallest item in I (not counting duplicates) occurs q_i times in I. Then the **maximum finger bound** for I, defined as $MF_I = \sum_{i=1}^{u} q_i \cdot (\log \min(i, u + 1 - i) + 1)$, satisfies $MF_I \in \Omega(H)$ where $H = \sum_{i=1}^{u} \left(q_i \cdot \log \frac{n}{q_i} \right)$.*

▶ **Definition 15** (Parallel Entropy-Sort). *Define a **bundle** of an item x to be a BT (binary tree) in which every leaf has a tagged copy of x. Let $PESort$ be the parallel merge-sort variant that does the following on an input batch I of items (I has subtrees $I.left$ and $I.right$):*

If $I.size \leq 1$, return I. Otherwise, compute $A = PEMerge(I.left)$ and $B = PEMerge(I.right)$ in parallel, and then parallel merge (Appendix A.2) A and B into an item-sorted batch C of bundles, combining bundles of the same item into one by simply making them the child subtrees of a new bundle, and then return C.

Then $PESort(I)$ returns an item-sorted batch of bundles, with one bundle (of all the tagged copies) for each distinct item in I, and clearly each bundle has height at most $I.height$.

▶ **Theorem 16** ($PESort$ Costs). *$PESort$ sorts every sequence in S^n with item frequencies $q_{1..u}$ within $O(H + n)$ work and $O\left((\log n)^2\right)$ span, where $H = \sum_{i=1}^{u} \left(q_i \cdot \ln \frac{n}{q_i} \right)$.*

Wait-Free Solvability of Equality Negation Tasks

Éric Goubault
École Polytechnique, Palaiseau, France
eric.goubault@lix.polytechnique.fr

Marijana Lazić
TU München, Munich, Germany
lazic@in.tum.de

Jérémy Ledent
École Polytechnique, Palaiseau, France
jeremy.ledent@lix.polytechnique.fr

Sergio Rajsbaum
Instituto de Matemáticas, UNAM, Mexico City, Mexico
sergio.rajsbaum@gmail.com

Abstract

We introduce a family of tasks for n processes, as a generalization of the two process *equality negation* task of Lo and Hadzilacos (SICOMP 2000). Each process starts the computation with a private input value taken from a finite set of possible inputs. After communicating with the other processes using immediate snapshots, the process must decide on a binary output value, 0 or 1. The specification of the task is the following: in an execution, if the set of input values is large enough, the processes should agree on the same output; if the set of inputs is small enough, the processes should disagree; and in-between these two cases, any output is allowed. Formally, this specification depends on two threshold parameters k and ℓ, with $k < \ell$, indicating when the cardinality of the set of inputs becomes "small" or "large", respectively. We study the solvability of this task depending on those two parameters. First, we show that the task is solvable whenever $k + 2 \leq \ell$. For the remaining cases ($\ell = k + 1$), we use various combinatorial topology techniques to obtain two impossibility results: the task is unsolvable if either $k \leq n/2$ or $n - k$ is odd. The remaining cases are still open.

2012 ACM Subject Classification Theory of computation → Concurrency

Keywords and phrases Equality negation, distributed computability, combinatorial topology

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.21

Funding The authors were supported by DGA project "Validation of Autonomous Drones and Swarms of Drones" and the academic chair "Complex Systems Engineering" of Ecole Polytechnique-ENSTA-Télécom-Thalès-Dassault-Naval Group-DGA-FX-FDO-Fondation ParisTech, by the UNAM-PAPIIT project IN109917, by the France-Mexico Binational SEP-CONACYT-ANUIES-ECOS grant M12M01, by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme under grant agreement No 787367 (PaVeS), as well as by the Austrian Science Fund (FWF) through Doctoral College LogiCS (W1255-N23).

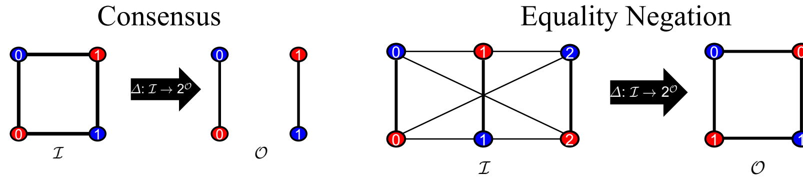
1 Introduction

The *equality negation* task is a variant of consensus that was defined by Lo and Hadzilacos [14], as the central idea to prove that the consensus hierarchy [10, 13] is not robust. Consider two processes P_0 and P_1, each of which has a private initial value, drawn from the set of possible input values $I = \{0, 1, 2\}$. Each process must irrevocably decide an output value either 0 or 1 so that the decisions of the processes are the same if and only if the initial values of the processes are different. It is well known that there is no wait-free consensus

© Éric Goubault, Marijana Lazić, Jérémy Ledent, and Sergio Rajsbaum;
licensed under Creative Commons License CC-BY

algorithm for two processes that uses only registers [6, 15]. The same is true for equality negation, because, as explained in [14], it is possible to solve consensus for two processes using an equality negation algorithm.

This result is intriguing for the following reason. It is well known that consensus is intimately related to connectedness. Namely, the reason why consensus is not wait-free solvable with registers is because its input complex is connected, while its specification requires deciding unto disconnected components of the output complex. The equality negation task is unsolvable for a different reason, since its output complex is connected. In our recent project [8], we have studied the unsolvability of this task, in the case of two processes, using methods from the field of episemic logic. The goal of this paper is to understand this task from a topological perspective. To do so, we extend it to the setting of more than two processes, and thus introduce a class of tasks which seem to be of a nature not previously studied.



Consider the following family of tasks, as a generalization of the equality negation problem, for n processes. These tasks are parametrized by two integers k, ℓ, with $1 \leq k < \ell \leq n$. In a given initial global state g of the system, each process has a private input value, and let $\mathit{InputSet}(g) \subseteq I$ be the set of these inputs, where I is the set of all possible input values. Thus, in one extreme, all processes may start with the same input value, and $|\mathit{InputSet}(g)| = 1$, and in the other they all start with different input values, and $|\mathit{InputSet}(g)| = n$. Similarly, let $\mathit{OutputSet}(g') \subseteq \{0, 1\}$ be the set of decided values in some final global state g' of the computation. For every $k, \ell \in \mathbb{N}$ such that $1 \leq k < \ell \leq n$ we define an equality negation task as follows, where g is any initial state, and g' is a final state of any execution starting with g:

- If $k < |\mathit{InputSet}(g)| < \ell$, the processes can decide any value in $\{0, 1\}$.
- If $|\mathit{InputSet}(g)| \geq \ell$, then $|\mathit{OutputSet}(g')| = 1$
- If $|\mathit{InputSet}(g)| \leq k$, then $|\mathit{OutputSet}(g')| > 1$

In this paper, we study the wait-free solvability of these tasks using read/write registers, depending on the two parameters k and ℓ. First, we show that if $\ell - k \geq 2$, the task is solvable. The remaining cases are the ones where $\ell = k + 1$ and $1 \leq k \leq n - 1$. In those cases, it is not possible anymore to have $k < |\mathit{InputSet}(g)| < \ell$: this means that there is no "gap" where the processes are free to decide any output without restriction. In order to prove that these remaining cases are unsolvable, we use various combinatorial topology techniques [11]. In particular, we show two impossibility results: the task is unsolvable if $k \leq n/2$ or if $n - k$ is odd. This leaves a few cases which are still open, namely, when $k > n/2$ and $n - k$ is even. We conjecture that they should also be unsolvable, but the right topological argument to prove it remains to be found.

Notice that the case of $n = 2$ processes is special, because the only option is that $k = 1$ and $\ell = 2$, in which case we obtain the equality negation task of Lo and Hadzilacos. Here the task is (trivially) solvable for $I = \{0, 1\}$, but becomes unsolvable if we add one more input value $I = \{0, 1, 2\}$. We discuss at the end of the paper why this case behaves differently.

Related work. The class of tasks that can be defined only in terms of sets of input values and corresponding sets of output values (without specifying which process gets which input nor which process should produce which output) are called *colorless tasks*. Since their introduction in [2], they have been thoroughly investigated; an overview and detailed citations can be found in [11]. There is a very elegant characterization of wait-free colorless tasks solvability, in terms of the existence of a certain simplicial map between the complex of possible input values and the complex of possible output values. The central colorless task is *set-agreement*, where each of n processes starts with an input, and each must halt with some process' input, such that no more than $n-1$ distinct inputs are chosen. The proof that set agreement is unsolvable using registers [12] revealed the deep connection between distributed computing and topology. Other examples of colorless tasks are consensus, loop agreement and approximate agreement.

Some tasks of interest are not colorless, and they are often much more difficult to analyze. For instance, the *renaming* task requires n processes, each starting with a unique input name, to choose distinct output from a range of size $2n-2$. This task can be solved if and only if n is not a prime power. The proof uses more involved algebraic topology techniques [3, 5] than the basic impossibility proof of set agreement based on Sperner's lemma.

More generally, colorless tasks seem to be about agreement, while others seem to be about symmetry breaking. Many such tasks have been considered, where processes only have their own (distinct) names as inputs. In *weak symmetry breaking*, the processes must choose binary outputs so that if all n processes participate, at least one chooses 0 and one chooses 1. In an *election* task, one process outputs 0 and exactly $n-1$ processes output 1. These and others are included in the *generalized symmetry breaking* family [4], defined by a set of possible output values, and for each value v, a lower bound and an upper bound on the number of processes that have to decide this value. These tasks are generally not only more complicated to analyze, but weaker than agreement tasks [4].

Equality negation tasks look very much like colorless tasks, in the sense that they are specified only according to the sets of input values of the processes, with no regard for which process has which value. But in fact, equality negation tasks are not colorless according to the definition in Section 4.1.4 of [11]. Indeed, the process names play an important role in the algorithms that we present to solve some instances of equality negation in Section 3.

2 Preliminaries

2.1 Equality negation tasks: from 2 to n processes

Note that the equality negation task defined in [14] for two processes does not have a unique generalization to the case with more than two processes.

In this paper we introduce a generalization that allows any number of input values, but we keep the set of possible decision values as $\{0,1\}$. It seems natural to define a generalization such that: (i) all the processes decide the same output value if they all have (pairwise) different input values, and (ii) they do not decide on the same output value if they all initially have the same input value. Still, this definition does not cover all possible combinations of input values of all processes, for example, if there are two processes with the same input, and a process with a different one. Therefore, there is a lot of freedom for defining output constraints for the remaining input cases. Our goal is to analyse all the possible generalizations of a certain form.

We define a family of equality negation tasks, based on the cardinality of the set of input values of all processes. Let n be the number of processes and let there exist exactly n input values, namely $I=\{0,1,2,\dots,n-1\}$. This assumption is done w.l.o.g., as we explain at the

end of the paper: all our results can be extended to $|I| > n$ in a straightforward manner. Every initial global state g defines the set of input values of all processes in the state g, that we denote by $\mathit{InputSet}(g)$. As this set is always a non-empty subset of the set I, its cardinality ranges from 1 to n. Note that the case when all processes have the same input value is represented by $|\mathit{InputSet}(g)| = 1$, and the case when all processes have different input values is the one when $|\mathit{InputSet}(g)| = n$. Different constraints on the intermediate cases introduce a family of tasks.

For every $k, \ell \in \mathbb{N}$ with $1 \leq k < \ell \leq n$ we define a generalized equality negation task:

- If in the initial global state g we have $|\mathit{InputSet}(g)| \geq \ell$, then all processes must decide the same value, either 0 or 1.
- If initially $|\mathit{InputSet}(g)| \leq k$, processes do not decide on the same value, i.e., there is at least one process deciding 0, and at least one deciding 1.
- If initially $k < |\mathit{InputSet}(g)| < \ell$, then each process can decide independently any value from $\{0, 1\}$.

The original equality negation task for two processes of Lo and Hadzilacos [14] is recovered as a special case, where $n = 2$, $I = \{0, 1, 2\}$, $k = 1$ and $\ell = 2$. It was shown to be unsolvable in the wait-free immediate shapshot model, by reduction to the consensus task. A more thorough study of this task for two processes, using topological methods and epistemic logic methods, can be found in [8].

2.2 Distributed computing

We consider the usual model consisting of n asynchronous processes that communicate through read/write shared registers of unbounded size [12]. When solving a task, initially, each process has some input value. After communicating with each other a finite number of times, each process produces an output value. The outputs produced are related to the inputs in the execution, as defined by the task specification.

In this paper, we do not consider crashing processes. All the n processes are present in the beginning of an execution, and they are guaranteed to run until the end of their program. Instead of crashes, the two ingredients that allow us to obtain impossibility results are asynchrony and wait-freedom. Although a bit unusual, this assumption is not restrictive: if we want to transpose our results to a model where the processes can crash, we can simply say that our task does not impose any restrictions on the outputs whenever at least one process has crashed.

The processes are executing in an *asynchronous* way, meaning that an execution consists of an arbitrary interleaving of the write and read operations of all the processes. Moreover, the program run by the processes is required to be *wait-free*: every process must be guaranteed to terminate its program in a bounded number of steps. Intuitively, these two restrictions mean that a process is not allowed to wait until it receives information from any of the other processes, since it may be arbitrarily slow.

We could describe our results in the general read/write registers model, but it is easier to do it in the *immediate snapshot* model. And we do so without loss of generality, because from the task computability perspective, the two models are known to be equivalent [1]. In the immediate snapshot model, each process has a designated memory cell where it can atomically write its input value. After doing so, it can atomically take a snapshot of the whole shared memory, in order to read the values that have been written by the other processes. Moreover, we restrict to a subset of all the executions described above: the snapshot is guaranteed to happen immediately after the write. For each pair of processes P and Q participating in this protocol, the immediate-snapshot may result in three possible outcomes:

- either P was faster than Q, in which case P did not see the value of Q, but Q saw P;
- or Q was faster than P, in which case the situation is reversed;
- or they executed concurrently, in which case they both saw each-other's input value.

Formally, an immediate snapshot execution consists of a sequence of sets of processes, where each set is called a *concurrency class*. All the processes in the same concurrency class will execute concurrently, and see each other's values, as well as the values of the previous concurrency classes.

These immediate snapshot executions give rise to simplicial complexes with nicer topological properties: namely, its effect on the input complex is to subdivide each simplex, thus preserving the topology. Immediate-snapshot operations can be wait-free implemented from read/write register operations, although at a quadratic cost in terms of the number of operations. For a survey including such results see e.g. [16].

2.3 Combinatorial topology

For our impossibility results, we use the combinatorial topology representation and basic results described in [11], briefly recalled here.

A pair (P_i, x_i) consisting of a process P_i along with its private (input or output) value x_i is called a *vertex*. An *input vertex* $v = (P_i, x_i)$ represents the initial state of process P_i, while an *output vertex* represents its decision at the end of a computation. A set σ of such vertices of cardinal $k+1$ is called a *k-simplex* or just *simplex*, and denotes the input values or output values of k distinct processes in an execution. It is also used to denote a global state in an execution, in which case x_i is the local state of P_i. If the x_i are input values, then σ is called an *input simplex*, if they are output values, it is an *output simplex*. A set of simplices closed under containment is called *simplicial complex*.

The *dimension* of a simplex σ, denoted $\dim(\sigma)$, is $|\sigma| - 1$, and it is *full* if it contains n vertices, one for each process. In a simplicial complex, a subset of a simplex, which is a simplex as well, is called a *face*. A simplex is *maximal* if it is not a strict subset of another simplex. A simplicial complex is *pure of dimension* $n-1$ if all its maximal simplices are full. The simplices of lower dimension are used in some distributed computing models to represent executions where some processes have crashed; but in our case, we will be mostly interested in the full simplices. The set of all possible input (resp. output) simplices forms an *input complex* $\mathcal{I}$ (resp. *output complex* $\mathcal{O}$).

A *coloring* of a complex is a mapping that assigns an element of a certain domain, usually called a color, to each vertex of the complex. A complex and its coloring form a *chromatic complex*. In distributed computing, coloring often assigns a distinct process identity to each vertex v of a simplex. We say that a simplex is *properly colored* (or it has proper coloring) if every two of its vertices are differently colored. We call a simplex *monochromatic* if all its vertices are labeled with the same color.

A *task* T for n processes is a triple $(\mathcal{I}, \mathcal{O}, \Delta)$ where $\mathcal{I}$ and $\mathcal{O}$ are pure chromatic $(n-1)$-dimensional complexes, such that every simplex is properly colored. The set of colors is $\{0, \dots, n-1\}$, and the colors are called *process names* or *ids*. The map Δ sends each simplex σ from $\mathcal{I}$ to a subcomplex $\Delta(\sigma)$ of $\mathcal{O}$. We say that Δ is a *carrier map* from the input complex $\mathcal{I}$ to the output complex $\mathcal{O}$. Briefly, the Asynchronous Computability Theorem [12] states that a task is wait-free solvable with registers if and only if there is a chromatic subdivision of the input complex and a chromatic simplicial map to the output complex, respecting the carrier map. Recall that a chromatic simplicial map sends simplices to simplices, preserving colors (process ids). We also use the fact that it can be assumed that a protocol solving a task using immediate snapshot operations induces an iterated chromatic subdivision of the input complex.

2.4 Index and Content of a Pseudomanifold

In this section, we introduce the main technical tool that we will use in Section 4.2.2 to prove impossibility results: the index lemma. The index lemma counts the properly colored n-simplices inside of a (not necessarily properly) colored and coherently oriented pseudomanifold, the *content*, by counting the properly colored $(n-1)$-simplices on the boundary, the *index*. The boundary determines the number of properly colored n-simplices in the interior of any pseudomanifold with that boundary.

Consider a set C with $n+1$ elements. A *sequence* S of C is an ordered list of the elements of C. We denote as S_i the *i-th* element of the sequence S, where $0 \leq i \leq n$. A *transposition* of S consists of interchanging the position of two elements S_i, S_j of S. If $i = j$ then we say that it is an *identity* transposition of S. A sequence S' of C is an *even transposition* of another sequence S if S' can be obtained with an even number of non identity transpositions over S. An *odd transposition* of S is defined similarly.

Let σ^n be a simplex with a proper coloring f with colors $ID^n = \{0, \ldots, n\}$. Consider a sequence S of ID^n. We say that S *induces* the sequence S' of σ^n with respect to f, when $f(S'_i) = S_i$, $0 \leq i \leq n$. If there is no ambiguity, we just say that S induces S'.

Let $\sigma^n = \{v_0, v_1, \ldots, v_n\}$ be a simplex. An *orientation* of σ^n is a set consisting of a sequence of its vertices and all even transpositions of this sequence. If $n > 0$ then there are exactly two possible orientations for σ^n: the sequence $\langle v_0, v_1, \ldots, v_n\rangle$ and all its even permutations, and the sequence $\langle v_1, v_0, v_2, \ldots, v_n\rangle$ and all its even permutations. For example, the two possible orientations of a 2-simplex can be seen as the clockwise and the counterclockwise directions, or the two possible orientations of a 1-simplex are the one from one of its vertices to the other, and the opposite direction. An orientation of σ^n, $n > 0$, *induces an orientation* on all of its $(n-1)$-faces: if σ_i^{n-1} is the $(n-1)$-face of σ^n without vertex v_i, then σ_i^{n-1} gets the same orientation if i is even, and otherwise it gets the opposite orientation. If σ^n, for example, is oriented $\langle v_0, v_1, \ldots, v_n\rangle$ then its face $\sigma_1^n = \langle v_0, v_2, v_3, \ldots, v_n\rangle$ gets orientation $\langle v_2, v_0, v_3, \ldots, v_n\rangle$, opposite of σ_1^n, as 1 is odd.

If σ^n has a proper coloring id with colors ID^n then we denote by $d = +1$ the orientation that contains the sequence induced by $\langle 0, 1, \ldots, n\rangle$, i.e., the sequence S of σ^n such that $id(S_i) = i$, $0 \leq i \leq n$, and denote by $d = -1$ the other orientation of σ^n which contains the sequence induced by $\langle 1, 0, \ldots, n\rangle$, the sequence S of σ^n such that $id(S_0) = 1$, $id(S_1) = 0$ and $id(S_i) = i$, $2 \leq i \leq n$. For $n = 0$, there is only one sequence of the vertices of a simplex σ^0, and then it has just one orientation, however we can associate $+1$ or -1 to this orientation. Hence, a 0-simplex also has two orientations. An orientation d of σ^n, $n > 0$, induces the orientation $(-1)^i d$ to σ_i^{n-1}, where σ_i^{n-1} is the $(n-1)$-face of σ^n without the vertex with id i.

A pseudomanifold K^n is *orientable* if there is an orientation for each of its n-simplices such that: if $\sigma, \sigma' \in K^n$ share a $(n-1)$-face τ then the two orientations on τ induced by σ and σ' are opposite. An orientation of K^n with this property is a *coherent orientation*. We say that K^n is *coherently oriented* if it has a coherent orientation. Let σ be an oriented n-simplex with a proper coloring c which uses colors ID^n. Consider a sequence S of ID^n. Let S' be the sequence of the vertices σ induced by S with respect to c. We say that S *belongs* to the orientation of σ with respect to c if S' belongs to the orientation of σ. The simplex σ is *counted by orientation* with respect to c in the following way: it is counted as $+1$ if the sequence $\langle 0, 1 \ldots n\rangle$ belongs to its orientation with respect to c, otherwise it is counted as -1. In the next definition, if $i \in ID^n$ is a color, we write $ID_i^n = ID^n \setminus \{i\}$.

▶ **Definition 1** (Index and Content). *Consider a coherently oriented pseudomanifold K^n with the induced orientation on its boundary $bd(K^n)$. Let c be a coloring, not necessarily proper, of K^n with ID^n.*

1. *The* content *of K^n, $C(K^n)$, with respect to c is the number of the properly colored n-simplices of K^n counted by orientation.*
2. *The* index *of K^n, $I_i(K^n)$, with respect to c is the number of properly colored $(n-1)$-simplices of $bd(K^n)$ with ID^n_i counted by orientation.*

If there is no ambiguity we simply write C^n or I^n_i. The next lemma is the restatement of Corollary 2 in [7] using our notation, and [9, pp. 46-47] for a simple version of dimension 2.

▶ **Lemma 2** (Index Lemma). *Let K^n be a coherently oriented pseudomanifold colored with ID^n. Then $C^n = (-1)^i I^n_i$.*

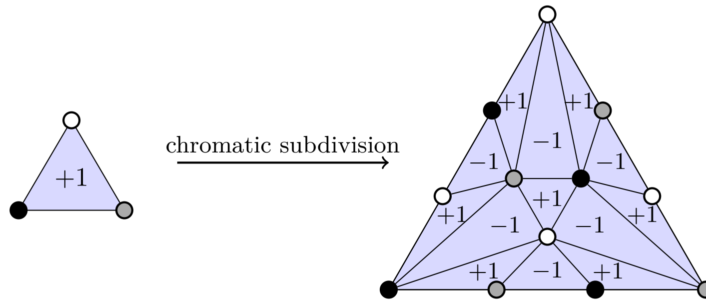
The coloring c of K^n is a simplicial map from $bd(K^n)$ to the boundary of a properly colored n-simplex σ^n with ID^n. Thus, we can think of the content of K^n as the number of times that $bd(K^n)$ is *wrapped around* $bd(\sigma^n)$, i.e., a combinatorial version of the notion of *degree* in topology.

To compute the index and content of chromatic subdivisions, we can just compute it from the original complex before subdivision happens. Indeed, we have the following Lemma:

▶ **Lemma 3.** *Let σ^n be a properly colored n-simplex with colors ID^n. Let $\chi(\sigma^n)$ be a chromatic subdivision of σ^n. Then $C(\chi(\sigma^n)) = C(\sigma^n)$.*

Proof. Assume w.l.o.g. that σ^n is counted positively. Thus, $C(\sigma^n) = 1$. We proceed by induction on n. For $n = 0$ (σ^0 is a single vertex), chromatic subdivisions do nothing so the result trivially holds. Now assume σ^n is of higher dimension.

By the index lemma, $C^n(\chi(\sigma^n)) = (-1)^n\, I^n_n(\chi(\sigma^n))$. Note that $ID^n_n = ID^{n-1}$. Let f be the $(n-1)$-face of σ^n with colors ID^{n-1}. Since $\chi(\sigma^n)$ is a chromatic subdivision of σ^n, the set K of properly colored $(n-1)$-simplexes on the boundary of $\chi(\sigma^n)$ with colors in ID^{n-1} is a chromatic subdivision of f. Thus, $I^n_n(\chi(\sigma^n)) = C^{n-1}(K) = C^{n-1}(f)$ by induction hypothesis, and $C^{n-1}(f) = I^n_n(\sigma^n) = (-1)^n\, C^n(\sigma^n)$. This concludes the proof. ◀



▶ **Corollary 4.** *Chromatic subdivisions preserve the index and content.*

3 Solvability analysis

If processes are not anonymous, we claim that the equality negation task, defined as above, is solvable if it holds that $\ell - k \geq 2$ (and $1 \leq k < \ell \leq n$).

▶ **Theorem 5.** *The algorithm from Listing 2 solves the equality-negation task if $\ell - k \geq 2$.*

Listing 1 Case $k = 1$ and $\ell = n$, for $n \geq 3$.

```
1  Boolean v := input_value
2
3  P_0: V := immediateSnapshot(v);
4      return 0;
5
6  P_i: V := immediateSnapshot(v);
7      if |V| ≥ 2 and |val(V)| ≤ 1 then
8        return 1
9      else return 0;
```

Listing 2 Arbitrary k and ℓ with $\ell - k \geq 2$.

```
1  Boolean v := input_value
2
3  P_0: V := immediateSnapshot(v);
4      return 0;
5
6  P_i: V := immediateSnapshot(v);
7      if |V| ≥ n+k−ℓ+1 and |val(V)| ≤ k then
8        return 1
9      else return 0;
```

Before we prove Theorem 5, and in order to understand the intuition behind it, let us first focus on its special case when $n \geq 3$, $k = 1$ and $\ell = n$, presented in Listing 1. In this figure, `immediateSnapshot(v)` returns the global state `V` of the system, `|V|` denotes the cardinal of the set `V` and `val(V)` denotes the set of local values that appear in the global state `V`. In this algorithm, one fixed process decides 0 independently of its input and the result of the snapshot. All the other processes decide depending on the output of the snapshot protocol. If a process sees at least 2 processes, and their input values are the same, the process decides 1. Otherwise it decides 0.

▶ **Proposition 6.** *The algorithm from Listing 1 solves the equality negation task if $n \geq 3$, $k = 1$ and $\ell = n$.*

Proof. As $k = 1$ and $\ell = n$, we need to show that (i) when all processes have the same input value, they will disagree, and (ii) when all processes have different inputs, then they all decide the same value (in this case 0).

Case (i). The distinguished process P_0 will decide 0, and thus we need to ensure that at least one process will decide 1. Using the properties of immediate snapshot, we know that among the processes P_i, $i \neq 0$, there is at least one process P_j that sees at least $n - 1$ processes. As $k \geq 1$ and $n \geq \ell \geq k + 2$, we have that $n \geq 3$, and thus $n - 1 \geq 2$. Hence, the processes P_j sees $n - 1 \geq 2$ processes, and they must have the same input value by the initial assumption of the case (i). Thus, P_j decides 1, which is enough for the disagreement.

Case (ii). Next we discuss the second requirement, that is, when no two processes have the same input value. We prove that all processes in this case agree and decide 0. Suppose by contradiction that a process P_i decides 1. According to the algorithm, this can happen only if $i \neq 0$, and if P_i has seen 2 processes with the same input value. This contradicts the initial assumption of this case. Hence, all processes decide 0, which concludes the proof. ◀

Let us now return to the general case, for any n, k and ℓ with $\ell - k \geq 2$ and $1 \leq k < \ell \leq n$, given in Listing 2. Similarly as in Listing 1, we fix one process that decides 0 independently of its input and the result of the snapshot. All the other processes decide depending on the output of the snapshot protocol. If a process sees at least $n + k - \ell + 1$ processes, and among their input values there are at most k different values, the process decides 1, and otherwise it decides 0. Now we are ready to prove Theorem 5.

Proof of Theorem 5. We prove that the algorithm from Listing 2 indeed solves equality-negation task analogously as in the case with $k = 1$ and $\ell = n$. We need to prove that (i) if $|InputSet(g)| \leq k$ for an initial state g, then at least one process decides 0 and at least one decides 1, and (ii) if $|InputSet(g)| \geq \ell$, then all processes decide the same value, here 0.

In the case (i) again we have process P_0 that decides 0 and we need to show that there is at least one process that decides 1. Similarly as in the special case, we know there is a process P_j, $j \neq 0$ that sees at least $n-1$ processes. As $\ell - k \geq 2$, we have that $n - 1 \geq n + k - \ell + 1$, and thus P_j has seen at least $n + k - \ell + 1$ processes and the set of their input values must have cardinality at most k by the assumption of the case (i). Hence, P_j decides 1.

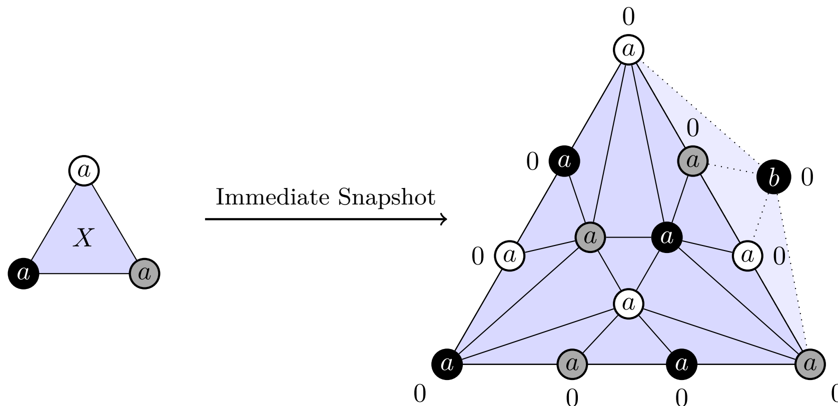
In the case (ii), initially there are at least ℓ different values. We want to prove that no process will decide 1. By means of contradiction, suppose that a process P_i decides 1. According to the algorithm, we have that $i \neq 0$ and P_i has seen at least $n + k - \ell + 1$ processes with at most k different input values. Note that P_i did not see any process with the remaining values, which are at least $\ell - k$, and that there must be at least one process with each of those values. Thus, we have at least $n + k - \ell + 1$ processes that P_i has seen and at least $\ell - k$ processes that P_i did not see, which is in total $n + 1$. As the total number of processes is n, this is a contradiction. This concludes the proof. ◀

4 Unsolvable cases

In the previous section, we have proved that whenever $\ell - k \geq 2$, the corresponding equality negation task is solvable. The remaining cases are the ones where $\ell = k + 1$, i.e., when there is no "gap" where the processes are allowed to decide any output value without constraints. We will show that most of the remaining cases are unsolvable, namely, when $k \leq n/2$ or when $n - k$ is odd. In the rest of the paper, we drop the ℓ parameter, since it will always be equal to $k + 1$. For these remaining cases, we provide partial results. First we give a simple argument to show that the task is unsolvable whenever $k \leq n/2$. Then, for the other values of k, we show that the task is unsolvable if $n - k$ is odd. The remaining cases are still open.

4.1 Impossibility proof when k is small

For simplicity, we first look at the case where $k = 1$; we will see later that it can be easily generalized to any $k \leq n/2$. For $k = 1$, the goal of the task is the following. If all the input values are the same, then the processes should disagree (i.e., not all of them should decide the same output). In all other cases, the processes should agree.



Let us assume for contradiction that the task is solvable in the immediate snapshot model. We focus on what happens in one of the initial global states where all the processes have the same input value a. Let us call $X = \{(P_0, a), \dots, (P_{n-1}, a)\}$ the corresponding simplex of the input complex. After the processes exchange information using iterated immediate snapshot communications, we obtain in the protocol complex a chromatic subdivision $\chi(X)$ of the original simplex X. The situation for 3 processes and one round of immediate snapshot is depicted above. The color of a vertex represents the process name; the value written inside a vertex is its input value; and next to it is the decision value.

In the input complex, the simplex X is surrounded by other simplices where not all inputs are the same. Thus, after the immediate snapshot computation occurs, the boundary of the subdivided simplex $\chi(X)$ is still surrounded by simplices with a different input value (three of them are depicted above, with input value b). In these simplices, the task specifies that all the processes must decide the same output. Assume w.l.o.g. that this output is 0. Since the boundary of $\chi(X)$ is connected, we can propagate the 0's step by step and we obtain that every vertex on the boundary of $\chi(X)$ must decide value 0.

To reach a contradiction, we will show that, given any assignment of output values 0 or 1 to the inner vertices of $\chi(X)$, there will always be at least one simplex of $\chi(X)$ where all the outputs are equal. This contradicts the task specification. Note that the next part of the proof only works because $\chi(X)$ is a *chromatic* subdivision; the statement does not hold for other subdivisions. If we regard the output values 0 and 1 as colors, the result that we want to prove looks like Sperner's lemma, where instead of counting the properly colored simplices inside the subdivision, we want to count the monochromatic ones, with respect to output values 0 and 1. To be able to use Sperner's lemma in this situation, we use a recoloring trick which was already used in [5] to obtain lower bounds on the renaming problem.

Suppose we are given an assignment of output values to the vertices of $\chi(X)$, such that all the vertices on the boundary decide 0. For v a vertex of $\chi(X)$, we write $d_v \in \{0, 1\}$ the decision value of v, and $\mathrm{id}_v \in \{0, \dots, n-1\}$ its process number. We define the recoloring of v to be $c_v := (\mathrm{id}_v + d_v) \mod n$. We have the following property:

▶ **Lemma 7.** *A simplex σ of $\chi(X)$ is monochromatic w.r.t. the decision values d_v if and only if it is proper w.r.t. the recoloring c_v.*

Proof. Since $\chi(X)$ is a chromatic subdivision of the simplex X, every simplex σ of $\chi(X)$ is properly colored w.r.t. the process numbers id_v. Thus, if σ is monochromatic w.r.t. the decision values d_v (that is, if all its vertices have the same decision value d), it is clear that the recoloring $c_v = (id_v + d) \mod n$ will still be proper. Conversely, we proceed by contraposition: suppose that not all decision values are the same. Then, there must be two vertices v and w of σ, such that $\mathrm{id}_w = (id_v + 1) \mod n$ and $d_v = 1$ and $d_w = 0$. Thus $c_v = c_w$, and the recoloring c is not proper. ◀

We can now easily conclude the proof. Since on the boundary of $\chi(X)$ all the vertices have the decision value 0, the recoloring c is just the process number: $c_v = \mathrm{id}_v$ for every boundary vertex v. Moreover, since $\chi(X)$ is just the (iterated) chromatic subdivision of a single input simplex X, we know that the process numbers id_v form a Sperner coloring on its boundary. Thus, by Sperner's lemma [11], there must exist a simplex $\sigma \in \chi(X)$ which is properly colored w.r.t. the recoloring c, and by Lemma 7 this means that all the vertices of this simplex decide the same output. Therefore, the equality negation task with parameter $k = 1$ cannot be solved by iterated immediate snapshot.

The same proof can be adapted for any $k \leq n/2$:

▶ **Theorem 8.** *The equality negation task for $n \geq 3$ processes, with parameters $k \leq n/2$ and $\ell = k + 1$ is not solvable in the immediate snapshot model.*

Proof. The only difference with the case $k = 1$ above is that we now start with the simplex $X = \{(P_0, 0), \dots, (P_{k-1}, k-1), (P_k, 0), \dots, (P_{2k-1}, k-1), \dots\}$, which has exactly k distinct input values, and every input appears at least twice. Therefore, if we remove one vertex from X, we obtain a face of X (of cardinality $n - 1$) which still has k distinct inputs. Then, this face belongs to another simplex with $k + 1$ input values. Thus, after subdivision, every process on the boundary of X must decide the same output (say, 0). Now the rest of the proof is exactly the same as in the case of $k = 1$: we have a simplicial complex $\chi(X)$ which is a chromatic subdivision of X, and we know that every process on its boundary has decision value 0. By Sperner's lemma, there is a simplex $\sigma \in \chi(X)$ which is properly colored w.r.t. the recoloring c, and by Lemma 7 it is monochromatic w.r.t. the decision values. Since X has $\leq k$ distinct input values, this set of outputs is not allowed. ◀

4.2 Some impossibility results depending on the parity of $n - k$

We now extend the proof of Section 4.1 to show that the task is not solvable by iterated immediate snapshot in half of the remaining cases: namely, whenever $n - k$ is odd, the task is unsolvable. We will rely on the same recoloring trick (and the associated Lemma 7), but instead of Sperner's lemma, we will use a slightly more powerful combinatorial topology tool called the *index lemma*. Sperner's lemma is equivalent to Brouwer's fixed point theorem which in turn, can be seen as reducing to distinguishing between the topology of a ball of dimension n with its boundary, the $(n-1)$-dimensional sphere. In more details, their topology is essentially different since there cannot be a continuous map from the n-ball *onto* (meaning, surjective) the $(n-1)$-sphere. The index lemma relates to a more subtle topological distinction, about the inexistence of continuous maps that would be winding around holes in a non-consistent way, so is not just about the image of the map. Sperner's lemma can be recovered as a direct consequence of the index lemma.

4.2.1 Low-dimensional example

Before we proceed to the general construction, let us illustrate the idea of the proof using the particular case of 3 processes and parameter $k = 2$. This instance of the task is the following: if the three input values are different, then the processes should decide the same output; otherwise, they should disagree. The input complex $\mathcal{I}$ contains every possible assignment of input values $\{0, 1, 2\}$ to the three processes $\{P, Q, R\}$. More formally, its maximal simplices are of the form $\sigma_{ijk} = \{(P, i), (Q, j), (R, k)\}$, for all $i, j, k \in \{0, 1, 2\}$. We can decompose this input complex $\mathcal{I}$ into two parts (depicted in Figure 1 below):

- There are 6 maximal simplices σ_{ijk} such that $|\{i, j, k\}| = 3$, that is, where all three processes have distinct values. In each of these simplices, the processes should agree on a common output value; moreover, since this subcomplex is connected, this common output has to be the same in all six of them.
- The rest of the input complex consists of simplices σ_{ijk} such that $|\{i, j, k\}| \leq 2$. In that part of the input, the processes should not agree. This part of the input complex is topologically a "pearl necklace" of three spheres.

Note that the 6 simplices with three distinct inputs are actually filling the hole in the middle of the "necklace". Thus, $\mathcal{I}$ is homotopy equivalent to a wedge of 2-spheres; in the terminology of [11], it is a *pseudosphere*.

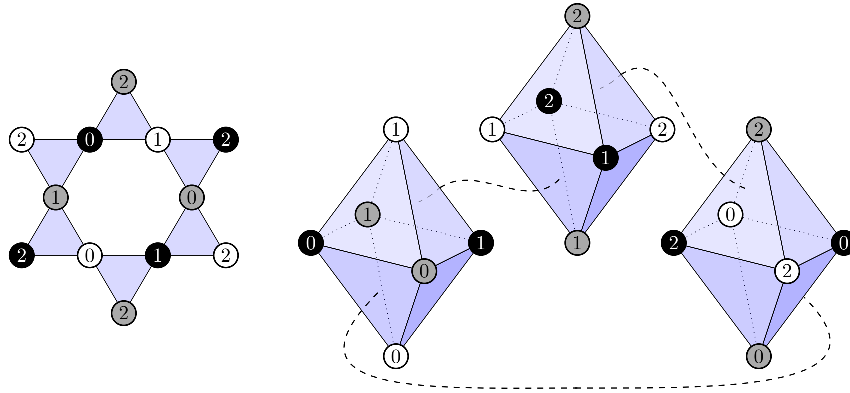


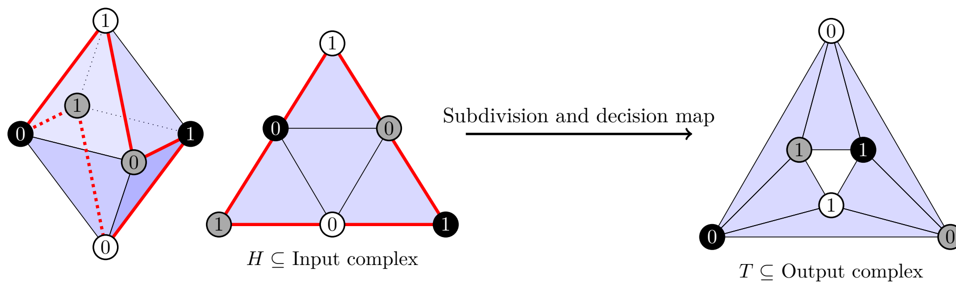
Figure 1 Exploded view of the input complex for $n = 3$ processes. All vertices with the same color and input value should be identified.

We now focus on one of the three spheres depicted in Figure 1; for example, the one where all inputs are either 0 or 1. That sphere has six edges which are labeled with two distinct input values 0 and 1. Each of these 01-edges also belongs to one of the six simplices with three inputs $0, 1, 2$: therefore, on each of these edges, all processes have to decide the same output value (say, w.l.o.g., that the common output value is 0). In the picture below, the 01-edges of the sphere are depicted in red. They form a circle on the surface of the sphere, splitting it in two half-spheres. We now look at only one of those two half-spheres, and call it H. It consists of four simplices, and its boundary contains only 01-edges. Thus, after the immediate snapshot computation occurs, this complex will be subdivided, and in order to solve the task, we should satisfy the following two conditions:

(1) All the vertices on the (red) boundary must be mapped to the same output, say, 0; and

(2) In every maximal simplex, not all processes should decide the same output.

This means we should have a simplicial map from a chromatic subdivision of H to the subcomplex T of the output complex where not all decision values are the same.



Since the processes on the boundary of H all decide output 0, the boundary of H has to be mapped to the outside boundary of T. Therefore, it seems clear topologically that some simplex inside H will necessarily be sent to the "111-hole" in the middle of T, contradicting condition (2). Notice however that the boundary of H is winding *twice* around the boundary of T. Moreover, this winding number is preserved by the chromatic subdivisions of H induced by the immediate snapshot protocol. Sperner's lemma, that we used in Section 4.1, only works if the boundaries are matched exactly (i.e., if we have a Sperner coloring). That's why we need to use the index lemma, which is able to handle cases where one boundary is winding several times around the other.

▶ **Proposition 9.** *The equality negation task for $n = 3$ processes, with parameters $k = 2$ and $\ell = 3$ is not solvable in the immediate snapshot model.*

Proof. Assume for contradiction that the task is solvable, and let H be the subcomplex of the input complex described above. So, there should be a chromatic subdivision $\chi(H)$ of H, and an assignment of decision values to its vertices, which solves the task. Remember that all the vertices on the boundary of $\chi(H)$ must decide the same output value, for example 0. We choose an arbitrary (coherent) orientation for the simplices of H; and we assign colors using the same recoloring as in Section 4.1. What we want to do now is prove that the content of $\chi(H)$ is non-zero. Using the index lemma, we can also compute its index. And by Corollary 4, applied to the boundary of $\chi(H)$, it is equal to the index of H itself.

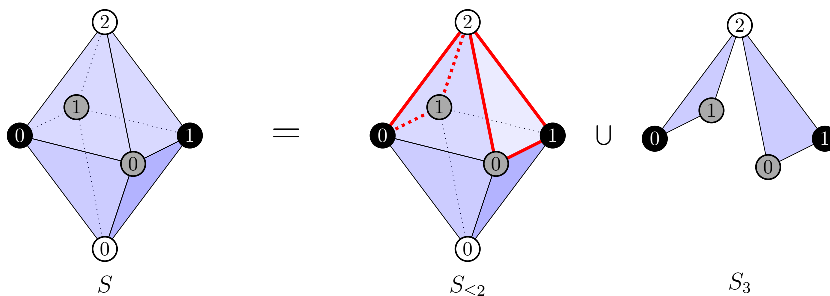
Then, an easy calculation shows that the index of H is either 2 or -2 depending on the orientation that we chose. So, no matter how many rounds of immediate snapshot we do, the index of $\chi(H)$ is either 2 or -2. By the index lemma, its content must be equal to its index (in absolute value). Since it is non-zero, there must exist proper triangles in $\chi(H)$, which by Lemma 7 correspond to monochromatic triangles w.r.t. the decision values. This contradicts the solvability of the task. ◀

4.2.2 General case

We now work with any k and n (remember however that $k < n$ since $\ell = k + 1 \leq n$), and we try to follow the same recipe as in the previous section. So, our goal is to find a subcomplex of the input complex, which is a pseudomanifold (in order to be able to apply the index lemma), and whose boundary consists of simplices with exactly k distinct input values (so that we can say that every process on the boundary has to decide the same output). A simple way to obtain a pseudomanifold is to choose a sphere in the input complex, and restrict ourselves to a subcomplex of this sphere. Although this idea is inspired from what we did in the low-dimensional example, the reader should be warned that the general construction we are about to do, when instantiated with $n = 3$ and $k = 2$, does not give the same proof as the one of Section 4.2.1.

Consider the subcomplex S of the input complex, which contains all the vertices of the form (P_i, a_i), where $a_i = 0$ or $a_i = i$ if $1 \leq i \leq k$, and $a_i = 0$ or $a_i = 1$ for all other values of i. The simplices of S are all combinations of such vertices, which are properly colored w.r.t. the process names. For example, $\sigma = \{(P_0, 0), (P_1, 0), \ldots, (P_{n-1}, 0)\}$ is a simplex of S. For ease of notation, when talking about maximal simplices, we omit the process names and just write $\sigma = \langle 0, \ldots, 0\rangle$, where the i-th component is the value of process P_i. Another simplex of S is $\langle 0, 1, 2, \ldots, k, 0, \ldots, 0\rangle$.

For the case $n = 3$, $k = 2$, the sphere $S = \{\langle a, b, c\rangle \mid a, b \in \{0, 1\}, c \in \{0, 2\}\}$ is represented below (the colors black, gray, white correspond to P_0, P_1, P_2, respectively).



Since every process can take exactly two different input values, independently from the other processes, the complex S is a $(n-1)$-sphere. In particular, it is a pseudomanifold. Among the maximal simplices of S, some of them contain exactly $k+1$ distinct input values, and the others contain k values or fewer. We write $S_{k+1} \subseteq S$ for the subcomplex of S which contains all the simplices with $k+1$ input values, and $S_{\leq k} = S \setminus S_{k+1}$. Both S_{k+1} and $S_{\leq k}$ are pseudomanifolds, and they have the same boundary $\partial S_{k+1} = \partial S_{\leq k} = S_{k+1} \cap S_{\leq k}$. In every simplex of S_{k+1}, the processes must decide the same value (since S_{k+1} is connected); assume w.l.o.g. that they decide the output value 0.

The complex $S_{\leq k}$ will play the role of the complex H that we had in Section 4.2.1. In the case of $n = 3$, $k = 2$, it corresponds to the sphere S with two holes corresponding to the two simplices with three distinct inputs. Its boundary is represented in red in the picture above. So the situation is a bit different than what we had in Section 4.2.1: instead of one boundary winding twice around the output, we now have two holes, each of them winding once around the output complex. Fortunately, the index lemma can also deal with such cases, as long as the two holes are winding in the same direction around the output (otherwise they would cancel each other). This is taken into account when we compute the index: here, one can check that the two holes have the same orientation, so the index will be either 2 or -2. For general k and n, the boundary of $S_{\leq k}$ can have many holes, each of them winding several times around the output complex. Our goal is once again to prove that the index is non-zero.

▶ **Theorem 10.** *The equality negation task for $n \geq 3$ processes, with parameters k and $\ell = k+1$ such that $n-k$ is odd, is not solvable in the immediate snapshot model.*

Proof. After the immediate snapshot communication occurs, we obtain a chromatic subdivision $\chi(S_{\leq k})$. We already know that it is a pseudomanifold, and that on its boundary every process has to decide 0. Our goal is now to use the index lemma to prove that there exists a simplex inside $\chi(S_{\leq k})$ which is monochromatic w.r.t. the decision values. First, we use the recoloring of Lemma 7, so that we are now searching for a properly colored simplex in $\chi(S_{\leq k})$. All we need to show is that the content of $\chi(S_{\leq k})$ is non-zero; thus, we want to compute its index. Since the index is preserved by chromatic subdivisions (see Corollary 4), we just need to compute the index of $S_{\leq k}$. Since the boundary of $S_{\leq k}$ is the same as the boundary of S_{k+1}, their index is the same (in absolute value), and since the index of S_{k+1} is equal to its content, we want to compute the content of S_{k+1}. In S_{k+1}, every simplex is properly colored (because everyone decides 0), so we just need to count the simplices of S_{k+1} by orientation. If the result is non-zero, this concludes the proof.

So let us pick an arbitrary orientation, say that the simplex $\langle 0, \dots, 0 \rangle$ is oriented positively. Each time we change the value of one coordinate, the orientation changes: for example, the simplex $\langle 1, 0, \dots, 0 \rangle$ is oriented negatively. Let us first characterize the maximal simplices of S_{k+1}: they are of the form $\langle a_0, a_1, 2, 3, \dots, k, a_{k+1}, \dots, a_{n-1} \rangle$, where for each $i \notin \{2, \dots, k\}$, $a_i \in \{0, 1\}$, and at least one of the a_i must be 0, and at least one must be 1. There are exactly $2^{n-k+1} - 2$ such simplices: all combinations of 0's and 1's are possible, except for $z = \langle 0, 0, 2, 3, \dots, k, 0, \dots, 0 \rangle$ and $u = \langle 1, 1, 2, 3, \dots, k, 1, \dots, 1 \rangle$. To go from z to u, we need to change $n-k+1$ coordinates. Thus, since $n-k+1$ is even, then z and u have the same orientation; otherwise, they would have opposite orientations.

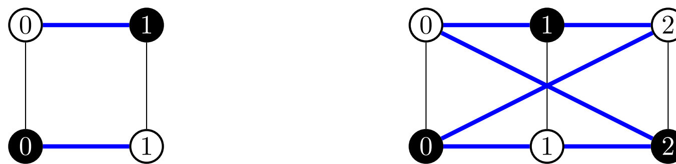
A simple calculation shows that summing the orientations of all the simplices of S_{k+1}, plus z and u, gives 0 as a result. Indeed, if we omit the middle components "$2, 3, \dots, k$", these 2^{n-k+1} simplices correspond to all the binary sequences of size $n-k+1$. We can, for example, enumerate those sequences using Gray code, so that the orientations are alternating, hence there is the same number of positively and negatively oriented simplices. Since z and u

are not in S_{k+1}, and they have the same orientation, then the content of S_{k+1} must be either 2 or -2. In any case, it is non-zero: so the task is not solvable whenever $n-k$ is odd. ◀

In particular, the case of $k = n-1$ is not solvable for any n, as $n-k = 1$ is odd. Note that in this proof, we do not really use the full power of the index lemma: we use it to show that the content of $S_{\leq k}$ is equal (in absolute value) to the content of S_{k+1}. This also follows from the simple fact that the content of a sphere is always 0, which is a direct consequence of the index lemma; so the contents of $S_{\leq k}$ and S_{k+1} must be opposite.

5 Discussion on the number of input values

In all this paper, we have been working with a set of input values $I = \{0, \dots, n-1\}$ of size n, for a number $n \geq 3$ of processes. This might seem a bit surprising, considering that in the original equality negation task for two processes [14], the size of the input set matters a lot. For $I = \{0, 1\}$, the task is solvable, but for $I = \{0, 1, 2\}$, it becomes unsolvable. It is quite informative to think about why our unsolvability proofs of Theorems 8 and 10 fail in the case of two processes and $I = \{0, 1, 2\}$. Both of them fail for a similar reason. We identify a subcomplex of the input complex ($\chi(X)$ in Section 4.1 and $S_{\leq k}$ in Section 4.2.2), and we surround its boundary by simplexes where every process must decide the same output. However, in order to assume that every vertex on the boudary decides the same output 0, we need to know that the part of the input complex which decides the same output is connected. For two processes, this is not the case if $I = \{0, 1\}$, however taking $I = \{0, 1, 2\}$ fixes this issue. Both input complexes are depicted below; the subcomplex where decisions should be the same is represented in blue.



No such problem occurs when there are more than 3 processes. Nevertheless, we can still wonder what happens when we allow an input set I of size $|I| > n$. Actually, all the results of this paper can be extended to such a setting in a straightforward way:

- Theorem 5 can easily be shown to work when more than n input values are allowed: intuitively, the algorithm relies only on the size of the sets of values that are seen by the processes. It never looks at the actual values.
- In all our unsolvability proofs (Theorems 8 and 10, and Proposition 9), we proceed by picking a subcomplex of the input complex, and then we work in this subcomplex to find a contradiction. If we add more input values, this just makes the input complex bigger, but all those proofs still work as they stand.

6 Conclusion

We have defined a family of equality negation tasks and studied their solvability and unsolvability. A few cases remain open questions; we conjecture that they should be unsolvable. The same proof method as in Section 4.2.2 using the index lemma might work for the remaining cases, but we would need to find another subcomplex of the input complex on which to apply it. Unfortunately, our attempts to do so have failed.

There are a number of other variations of this task that we could have studied. For example, instead of having binary outputs in $\{0,1\}$, we could have a larger set of output values. Then, we could introduce two more parameters to define what it means to "agree" or to "disagree" in that context. That kind of parameters appear in the generalized symmetry breaking task [4].

References

1. H. Attiya and S. Rajsbaum. The Combinatorial Structure of Wait-Free Solvable Tasks. *SIAM J. Comput.*, 31(4):1286–1313, 2002. doi:10.1137/S0097539797330689.
2. E. Borowsky, E. Gafni, N. Lynch, and S. Rajsbaum. The BG Distributed Simulation Algorithm. *Distrib. Comput.*, 14(3):127–146, October 2001. doi:10.1007/PL00008933.
3. Armando Castañeda, Maurice Herlihy, and Sergio Rajsbaum. An Equivariance Theorem with Applications to Renaming. *Algorithmica*, 70(2):171–194, October 2014. doi:10.1007/s00453-013-9855-3.
4. Armando Castañeda, Damien Imbs, Sergio Rajsbaum, and Michel Raynal. Generalized Symmetry Breaking Tasks and Nondeterminism in Concurrent Objects. *SIAM J. Comput.*, 45(2):379–414, 2016. doi:10.1137/130936828.
5. Armando Castañeda and Sergio Rajsbaum. New combinatorial topology bounds for renaming: the lower bound. *Distributed Computing*, 22(5):287–301, August 2010. doi:10.1007/s00446-010-0108-2.
6. Benny Chor, Amos Israeli, and Ming Li. On Processor Coordination Using Asynchronous Hardware. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, pages 86–97, New York, NY, USA, 1987. ACM. doi:10.1145/41840.41848.
7. Ky Fan. Simplicial maps from an orientable n-pseudomanifold into S^m with the octahedral triangulation. *Journal of Combinatorial Theory*, 2(4):588–602, 1967. doi:10.1016/S0021-9800(67)80063-2.
8. Éric Goubault, Marijana Lazić, Jérémy Ledent, and Sergio Rajsbaum. A Dynamic Epistemic Logic analysis of the Equality Negation task. *Dynamic Logic: New Trends and Applications, DaLi 2019*, to appear in Springer LNCS.
9. Michael Henle. *A Combinatorial Introduction to Topology.* Dover, 1983. doi:10.2307/1574757.
10. Maurice Herlihy. Wait-free Synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991. doi:10.1145/114005.102808.
11. Maurice Herlihy, Dmitry Kozlov, and Sergio Rajsbaum. *Distributed Computing Through Combinatorial Topology.* Morgan Kaufmann, 2013.
12. Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *J. ACM*, 46(6):858–923, 1999. doi:10.1145/331524.331529.
13. Prasad Jayanti. On the Robustness of Herlihy's Hierarchy. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Distributed Computing*, PODC '93, pages 145–157, New York, NY, USA, 1993. ACM. doi:10.1145/164051.164070.
14. Wai-Kau Lo and Vassos Hadzilacos. All of Us Are Smarter than Any of Us: Nondeterministic Wait-Free Hierarchies Are Not Robust. *SIAM J. Comput.*, 30(3):689–728, 2000. doi:10.1137/S0097539798335766.
15. Hosame H Abu-Amara Michael C Loui. Memory requirements for agreement among unreliable asynchronous processes. In *Advances in Computing research*, pages 163–183, Greenwich, CT, 1987. JAI Press.
16. Sergio Rajsbaum, Michel Raynal, and Panagiota Fatourou. An Introductory Tutorial to Concurrency-Related Distributed Recursion. *Bulletin of the EATCS*, 111, 2013. URL: http://eatcs.org/beatcs/index.php/beatcs/article/view/223.

Scalable Byzantine Reliable Broadcast

Rachid Guerraoui
EPFL, Lausanne, Switzerland
rachid.guerraoui@epfl.ch

Petr Kuznetsov
LTCI, Télécom Paris, IP Paris, Paris, France
petr.kuznetsov@telecom-paristech.fr

Matteo Monti
EPFL, Lausanne, Switzerland
matteo.monti@epfl.ch

Matej Pavlovic
EPFL, Lausanne, Switzerland
matej.pavlovic@epfl.ch

Dragos-Adrian Seredinschi
EPFL, Lausanne, Switzerland
dragos-adrian.seredinschi@epfl.ch

Abstract

Byzantine reliable broadcast is a powerful primitive that allows a set of processes to agree on a message from a designated sender, even if some processes (including the sender) are Byzantine. Existing broadcast protocols for this setting scale poorly, as they typically build on *quorum systems* with strong intersection guarantees, which results in linear per-process communication and computation complexity.

We generalize the Byzantine reliable broadcast abstraction to the probabilistic setting, allowing each of its properties to be violated with a fixed, arbitrarily small probability. We leverage these relaxed guarantees in a protocol where we replace quorums with stochastic *samples*. Compared to quorums, samples are significantly smaller in size, leading to a more scalable design. We obtain the first Byzantine reliable broadcast protocol with logarithmic per-process communication and computation complexity.

We conduct a complete and thorough analysis of our protocol, deriving bounds on the probability of each of its properties being compromised. During our analysis, we introduce a novel general technique that we call adversary decorators. Adversary decorators allow us to make claims about the optimal strategy of the Byzantine adversary without imposing any additional assumptions. We also introduce Threshold Contagion, a model of message propagation through a system with Byzantine processes. To the best of our knowledge, this is the first formal analysis of a probabilistic broadcast protocol in the Byzantine fault model. We show numerically that practically negligible failure probabilities can be achieved with realistic security parameters.

2012 ACM Subject Classification Mathematics of computing → Probabilistic algorithms; Mathematics of computing → Stochastic processes; Theory of computation → Distributed algorithms; Theory of computation → Distributed computing models

Keywords and phrases Byzantine reliable broadcast, probabilistic distributed algorithms, scalable distributed systems, stochastic processes

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.22

Related Version An extended version of this article – which includes a thorough analysis of our protocols – appears on arXiv at `https://arxiv.org/abs/1908.01738v1` [34].

Funding This work has been supported in part by the European ERC Grant 339539 - AOC.

© Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi;
licensed under Creative Commons License CC-BY

33rd International Symposium on Distributed Computing (DISC 2019).
Editor: Jukka Suomela; Article No. 22; pp. 22:1–22:16

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Broadcast is a popular abstraction in the distributed systems toolbox, allowing a process to transmit messages to a set of processes. The literature defines many flavors of broadcast, with different safety and liveness guarantees [14, 25, 35, 42, 48]. In this paper we focus on Byzantine reliable broadcast, as defined by Bracha [12]. This abstraction is a central building block in practical Byzantine fault-tolerant (BFT) systems [15, 19, 33]. We tackle the problem of its scalability, namely reducing the complexity of Byzantine reliable broadcast, and seeking good performance despite a large number of participating processes.

In Byzantine reliable broadcast, a designated sender broadcasts a single message. Intuitively, the broadcast abstraction ensures that no two correct processes deliver different messages (*consistency*), either all correct processes deliver a message or none does (*totality*), and that, if the sender is correct, all correct processes eventually deliver the broadcast message (*validity*). This must hold despite a certain fraction of Byzantine processes, potentially including the sender. We denote by N the number of processes in the system, and f the fraction of processes that are Byzantine. Existing algorithms for Byzantine reliable broadcast scale poorly as they typically have $O(N)$ per-process communication complexity [13, 42, 45, 53]. The root cause for poor scalability of these algorithms is their use of quorums [43, 56], i.e., sets of processes that are large enough to always intersect in at least one correct process. The size of a quorum grows linearly with the size of the system [14].

To overcome the scalability limitation of quorum-based broadcast, Malkhi *et al.* [46] generalize quorums to the probabilistic setting. In this setting, two random quorums intersect with a fixed, arbitrarily high probability, allowing the size of each quorum to be reduced to $O(\sqrt{N})$. We are not aware of any Byzantine reliable broadcast algorithm building on probabilistic quorums; nevertheless, such an algorithm could have a per-process communication complexity reduced from $O(N)$ to $O(\sqrt{N})$. The active_t protocol [42] uses a form of samples for an optimistic path, but relies on synchrony and has a linear worst-case complexity (that is arguably very likely to occur with merely a moderate number of faulty processes).

1.1 Samples

In this paper, we present a probabilistic gossip-based Byzantine reliable broadcast algorithm having $O(\log N)$ per-process communication and computation complexity, at the expense of $O(\log N/\log\log N)$ latency. Essentially, we propose *samples* as a replacement for quorums. Like a probabilistic quorum, a sample is a randomly selected set of processes. Unlike quorums, samples do not need to intersect. Samples can be significantly smaller than quorums, as each sample must be large enough only to be *representative* of the system with high probability.

A process can use its sample to gather information about the global state of the system. An old Italian saying provides an intuitive understanding of this shift of paradigm: *"To know if the sea is salty, one needs not drink all of it!"* Intuitively, we leverage the law of large numbers, trading performance for a fixed, arbitrarily small probability of non-representativeness. [1]

Throughout this paper, we extensively use samples to estimate the number of processes satisfying a set of *yes-or-no* predicates, e.g., the number of processes that are ready to deliver a message m. Consider the case where a correct process π queries K randomly selected

[1] To get an intuition of the difference between quorums and samples, consider the emulation of a shared memory in message passing [3]. One writes in a quorum and reads from a quorum to fetch the last value written. Our algorithms are rather in the vein of "write all, read any." Here we would "write" using a gossip primitive and "sample" the system to seek the last written value.

processes (a sample) for a predicate P. Assume that a fraction p of correct processes from the whole system satisfies predicate P. Let x be the fraction of positive responses (out of K) that π collects. By the Chernoff bound, the probability of $|x - p| \geq f + \epsilon$ is smaller or equal to $\exp(-\lambda(\epsilon)K)$, where λ quickly increases with ϵ. For sufficient K, the probability of x differing from p by more than $f + \epsilon$ can be made exponentially small.

Our algorithms use a *sampling oracle* that returns the identity of a process from the system picked with uniform probability. In a permissioned system (i.e., one where the set of participating processes is known) sampling reduces to picking with uniform probability an element from the set of processes. In a permissionless system subject to Byzantine failures and slow churn, a (nearly) uniform sampling mechanism is achievable using gossip [10].

1.2 Scalable Byzantine Reliable Broadcast

Our probabilistic algorithm, Contagion, allows each property of Byzantine reliable broadcast to be violated with an arbitrarily small probability ϵ. We show that ϵ scales sub-quadratically with N, and decays exponentially in the size of the samples. As a result, for a fixed value of ϵ, the per-node communication complexity of Contagion is logarithmic.

We build Contagion incrementally, relying on two sub-protocols, as we describe next.

First, **Murmur** is a probabilistic broadcast algorithm that uses simple message dissemination to establish *validity* and *totality*. In this algorithm, each correct process relays the sender's message to a randomly picked *gossip sample* of other processes. For the sample size $\Omega(\log N)$, the resulting gossip network is a connected graph with $O(\log N/\log\log N)$ diameter, with high probability [21, 17]. In case of a Byzantine sender, however, Murmur does not guarantee consistency.

Second, **Sieve** is a probabilistic consistent broadcast algorithm (built upon Murmur) that guarantees *consistency*, i.e., no two correct processes deliver different messages. To do so, each correct process uses a randomly selected *echo sample*. Intuitively, if enough processes from any echo sample confirm a message m, then with high probability no correct processes in the system delivers a different message m'. Sieve, however, does not ensure totality. If a Byzantine sender broadcasts multiple conflicting messages, a correct process might be unable to gather sufficient confirmations for either of them from its echo sample, and consequently would not deliver any message, even if some correct process delivers a message.

Finally, **Contagion** is a probabilistic Byzantine reliable broadcast algorithm that guarantees validity, consistency, and totality. The sender uses Sieve to disseminate a consistent message to a subset of the correct processes. In order to achieve totality, Contagion mimics the spreading of a contagious disease in a population. A process samples the system and if it observes enough other "infected" processes in its sample, it becomes infected itself. If a critical fraction of processes is initially infected by having received a message from the underlying Sieve layer, the message spreads to all correct processes with high probability. If a process observes enough other infected processes, it delivers. As in the original deterministic implementation by Bracha [12], the crucial point here is that "enough" for becoming infected is less than "enough" for delivering. This way, with high probability, either all correct processes deliver a message or none does – Contagion satisfies totality. The other two important properties (validity and consistency) are inherited from the underlying (Contagion and Sieve) layers.

1.3 Probability Analysis and Applications

A major technical contribution of this work is a complete, formal analysis of the properties of our three algorithms. To the best of our knowledge, this is the first such analysis applied to a probabilistic broadcast algorithm in the Byzantine fault model, and this turned out to be challenging. Intuitively, providing a bound on the probability of a property being violated reduces to studying a joint distribution between the inherent randomness of the system and the behavior of the Byzantine adversary. Since the behavior of the adversary is arbitrary, the marginal distribution of the Byzantine's behavior is unknown.

We develop two novel strategies to bound the probability of a property being violated, which we use in the analysis of Sieve and Contagion respectively.

(1) When evaluating the consistency of Sieve, we show that a bound holds for every possibly optimal adversarial strategy. Essentially, we identify a subset of adversarial strategies that we prove to include the optimal one, i.e., the one that has the highest probability of compromising the consistency of Sieve. We then prove that every possibly optimal adversarial strategy has a probability of compromising the consistency of Sieve smaller than some ϵ.

(2) When evaluating the totality of Contagion, we show that the adversarial strategy does not affect the outcome of the execution. Here, we show that any adversarial strategy reduces to a well-defined sequence of choices. We then prove that, due to the limited knowledge of the Byzantine adversary, every choice is equivalent to a random one.

Our analysis shows that, for a practical choice of parameters, the probability of violating the properties of our algorithm can be brought down to 10^{-16} for systems with thousands of processes.

In the rest of this paper, we state our system model and assumptions (Section 2), and then present our Murmur, Sieve, and Contagion algorithms (Sections 3 to 5). Our algorithm descriptions are high-level, but throughout this paper we will often refer the interested reader to the corresponding appendices containing all details (including pseudocode and formal proofs); to respect conference proceedings space limits, we place these appendices in the extended version of this article [34]. We discuss the security and complexity of our algorithms in Section 6, and cover related work in Section 7.

2 Model and Assumptions

We assume an asynchronous message-passing system where the set Π of $N = |\Pi|$ processes partaking in an algorithm is fixed. Any two processes can communicate via a reliable authenticated point-to-point link.

We assume that each correct process has access to a local, unbiased, independent source of randomness. We assume that every correct process has direct access to an oracle Ω that, provided with an integer $n \leq N$, yields the identities of n distinct processes, chosen uniformly at random from Π. Implementing Ω is beyond the scope of this paper, but it is straightforward in practice. In a system where the set of participating processes is known, sampling reduces to picking with uniform probability an element from the set of processes. In a system without a global membership view that may even be subject to slow churn, a (nearly) uniform sampling mechanism is available in literature due to Bortnikov *et al.* [10].

At most a fraction f of the processes are Byzantine, i.e., subject to arbitrary failures [40]. Byzantine processes may collude and coordinate their actions. Unless stated otherwise, we denote by $\Pi_C \subseteq \Pi$ the set of correct processes and by $C = |\Pi_C| = (1 - f)N$ the number of

correct processes. We assume a static Byzantine adversary controlling the faulty processes, i.e., the set of processes controlled by the adversary is fixed at the beginning and does not change throughout the execution of the protocols.

We make standard cryptographic assumptions regarding the power of the adversary, namely that it cannot subvert cryptographic primitives, e.g., forge a signature. We also assume that Byzantine processes are not aware of (1) the output of the local source of randomness of any correct process; and (2) which correct processes are communicating with each other. The latter assumption is important to prevent the adversary from poisoning the view of the system of a targeted correct process without having to bias the local randomness source of any correct process. Even against ISP-grade adversaries, we can implement this assumption in practice by means such as onion routing [18] or private messaging [54].

3 Probabilistic Broadcast with Murmur

In this section, we introduce the *probabilistic broadcast* abstraction and its implementation, Murmur. Briefly, probabilistic broadcast ensures validity and totality. We use this abstraction in Sieve (Section 4) to initially distribute the message from a sender to all correct processes.

The probabilistic broadcast interface assumes a specific sender process σ. An instance pb of probabilistic broadcast exports two events. First, process σ can request through $\langle pb.\text{Broadcast} \mid m\rangle$ to broadcast a message m. Second, the indication event $\langle pb.\text{Deliver} \mid m\rangle$ is an upcall for delivering message m broadcast by σ. For any $\epsilon \in [0, 1]$, we say that probabilistic broadcast is ϵ-secure if:

1. **No duplication:** No correct process delivers more than one message.
2. **Integrity:** If a correct process delivers a message m, and σ is correct, then m was previously broadcast by σ.
3. **ϵ-Validity:** If σ is correct, and σ broadcasts a message m, then σ eventually delivers m with probability at least $(1 - \epsilon)$.
4. **ϵ-Totality:** If a correct process delivers a message, then every correct process eventually delivers a message with probability at least $(1 - \epsilon)$.

3.1 Gossip-based Algorithm

Murmur (presented in detail in [34, Appendix A, Algorithm 1]) distributes a single message across the system by means of gossip: upon reception, a correct process relays the message to a set of randomly selected neighbors. The algorithm depends on one parameter: *expected gossip sample size* G.

Upon initialization, every correct process uses the sampling oracle Ω to select (on average) G other processes to gossip with. Gossip links are reciprocated, making the gossip graph undirected.

To broadcast a message m, the designated sender σ signs m and sends it to all its neighbors. Upon receiving a correctly signed message m from σ for the first time, each correct process delivers m and forwards m to every process in its neighborhood.

3.2 Analysis Using Erdös-Rényi Graphs

The detailed analysis, provided in [34, Appendix A, Sections A.3 and A.4], formally proves the correctness of Murmur by deriving a bound on ϵ as a function of the algorithm and system parameters. Here we give a very high-level sketch of our probabilistic analysis of Murmur.

3.2.1 No duplication, integrity and ϵ-validity

Here is the intuition behind the properties satisfied by Murmur [34, Appendix A.3]:

1. No duplication: A correct process maintains a *delivered* variable that it checks and updates when delivering a message, preventing it from delivering more than one message.
2. Integrity: Before broadcasting a message, the sender signs that message with its private key. Before delivering a message m, a correct process verifies m's signature. This prevents any correct process from delivering a message that was not previously broadcast by the sender.
3. ϵ-Validity: Upon broadcasting a message, the sender also immediately delivers it. Since this happens *deterministically*, Murmur satisfies 0-validity, independently from the parameter G.

3.2.2 ϵ-Totality

Murmur satisfies ϵ-totality with ϵ upper-bounded by a function that decays exponentially with G, and increases polynomially with f [34, Appendix A.4]. We prove that the network of connections established among the correct processes is an undirected Erdős–Rényi graph [21]. Totality is satisfied if such graph is connected.

Erdős–Rényi graphs are well known in literature [1] to display a connectivity phase transition: when the expected number of connections each node has exceeds the logarithm of the number of nodes, the probability of the graph being connected steeply increases from 0 to 1 (in the limit of infinitely large systems, this increase becomes a step function). We use this result to compute the probability of the sub-graph of correct processes being connected and, consequently, of Murmur satisfying totality ([34, Theorem 4]).

4 Probabilistic Consistent Broadcast with Sieve

In this section, we first introduce the probabilistic consistent broadcast abstraction, which allows (a subset of) the correct processes to agree on a single message from a (potentially Byzantine) designated sender. We then discuss Sieve, an implementation of this abstraction. We use probabilistic consistent broadcast in the implementation of Contagion (see Section 5) as a way to consistently disseminate messages. Sieve itself builds on top of probabilistic broadcast (see Section 3).

Probabilistic consistent broadcast does not guarantee totality, but it does guarantee consistency: despite a Byzantine sender, no two correct processes deliver different messages. If the sender is Byzantine, however, it may happen with a non-negligible probability that only a proper subset of the correct processes deliver the message.

For any $\epsilon \in [0, 1]$, we say that probabilistic consistent broadcast is ϵ-secure if it satisfies the properties of **No duplication** and **Integrity** as defined above, and:

- **ϵ-Total validity:** If σ is correct, and σ broadcasts a message m, every correct process eventually delivers m with probability at least $(1 - \epsilon)$.
- **ϵ-Consistency:** With probability at least $(1 - \epsilon)$, no two correct processes deliver different messages.

4.1 Sample-Based Algorithm

Sieve (presented in detail in [34, Appendix B, Algorithm 3]) uses Echo messages to consistently distribute a single message to (a subset of) the correct processes: before delivering a message, a correct process samples the system to estimate how many other processes received the same message. The algorithm depends on two parameters: the *echo sample size* E and the *delivery threshold* $\hat{E}$.

Upon initialization, every correct process uses the sampling oracle Ω to select an *echo sample* $\mathcal{E}$ of size E, and sends an EchoSubscribe message to every process in $\mathcal{E}$. Upon broadcasting, the sender uses the underlying probabilistic broadcast (e.g., Murmur) to initially distribute a message to every correct process. This step does not ensure consistency, so processes may see conflicting messages if the sender σ is Byzantine. Upon receiving a message m from probabilistic broadcast, a correct process π sends an $(\texttt{Echo}, m)$ message to every process that sent an EchoSubscribe message to π. (Note that, due to the no duplication property of probabilistic broadcast, this can happen only once per process.) Upon collecting $\hat{E}$ $(\texttt{Echo}, m)$ messages from its echo sample $\mathcal{E}$, π delivers m. Notably, if π delivers m, then with high probability every other correct process either also delivers m, or does not deliver anything at all, but never delivers $m' \neq m$.

4.2 Analysis Using Adversary Decorators

Here we present a high-level outline of the analysis of Sieve; for a complete formal treatment, see [34, Appendix B], where we prove the correctness of Sieve by deriving a bound on ϵ.

4.2.1 No duplication and integrity

Sieve deterministically satisfies these properties the same way as Murmur does [34, Appendix B.3].

4.2.2 ϵ-Total Validity

Since we assume a correct sender σ (by the premise of total validity), a bound on the probability ϵ of violating total validity can easily be derived from the probability of the underlying probabilistic broadcast failing and from the probability of some process' random echo sample having more than $E - \hat{E}$ Byzantine processes [34, Appendix B.4].

4.2.3 ϵ-Consistency

While the intuition why Sieve satisfies consistency is rather simple, proving it formally is the most technically involved part of this paper. We now provide the intuition and present the techniques we use to prove it, while deferring the full body of the formal proof to [34, Appendices B.5-B.10].

In order for Sieve to violate consistency, two correct processes must deliver two different messages (which can only happen if the sender σ is malicious). This, in turn, means that two correct processes π and π' must observe two different messages m and m' sufficiently represented in their respective echo samples. I.e., π receives $(\texttt{Echo}, m)$ at least $\hat{E}$ times and π' receives $(\texttt{Echo}, m')$ at least $\hat{E}$ times.

Note that a correct process only sends $(\texttt{Echo}, m)$ for a single message m received from the underlying probabilistic broadcast layer. The intuition of Sieve is the same as in quorum-based algorithms. With quorums, if enough correct processes issue $(\texttt{Echo}, m)$ to make at

least one correct process deliver m, the remaining processes (regardless of the behavior of the Byzantine ones) are not sufficient to make any other correct process deliver m'. For Sieve, this holds with high probability as long as $\hat{E}$ is sufficiently high and the fraction f of Byzantine processes is limited.

To prove these intuitions, we first describe Simplified Sieve [34, Appendix B.6], a strawman variant of Sieve that is easier to analyze. We prove that Simplified Sieve guarantees consistency with strictly lower probability than Sieve does [34, Appendix B.8, Lemma 12]. Thus, an upper bound on the probability of Simplified Sieve failing is also an upper bound on the probability of Sieve failing.

Next, we analyze Simplified Sieve using a novel technique that involves modeling the adversary as an algorithm that interacts with the system through a well-defined interface [34, Appendix B.7]. We start from the set of all possible adversarial algorithms and gradually reduce this set, while proving that the reduced set still includes an *optimal* adversary [34, Appendix B.9]. (An adversary is optimal if it maximizes the probability ϵ of violating consistency.) Intuitively, we prove that certain actions of the adversary always lead to strictly lowering ϵ, and thus need not be considered. For example, an adversary can only decrease its chance of compromising consistency when omitting Echo messages.

To this end, we introduce the concept of *decorators*. A decorator is an algorithm that lies between an adversary and a system. It emulates a system and exposes the corresponding interface to the decorated adversary. At the same time, the decorator also exposes the interface of an adversary to interact with a system. The purpose of a decorator is to alter the interaction between the adversary and the system. For any decorated adversary, we prove that the decorator does not decrease the probability ϵ of the adversary compromising the system. Thus, a decorator effectively transforms an adversary into a stronger one. Each decorator maps a set of adversaries into one of its proper subsets that is easier to analyze [34, Appendix D].

Through a series of decorators, we obtain a tractable set of adversaries that provably contains an optimal one. Then we derive the bound on ϵ under these adversaries [34, Theorem 9].

5 Probabilistic Byzantine Reliable Broadcast with Contagion

Our main algorithm, Contagion, implements the probabilistic Byzantine reliable broadcast abstraction. This abstraction is strictly stronger than probabilistic consistent broadcast, as it additionally guarantees ϵ-totality. Despite a Byzantine sender, either none or every correct process delivers the broadcast message.

For any $\epsilon \in [0, 1]$, we say that probabilistic Byzantine reliable broadcast is ϵ-secure if it satisfies the properties of **No duplication**, **Integrity**, **ϵ-Validity**, **ϵ-Consistency** and **ϵ-Totality**, as already defined in previous sections.

5.1 Feedback-Based Algorithm

Our algorithm implementing probabilistic Byzantine reliable broadcast is called Contagion and we present it in detail in [34, Appendix C, Algorithm 7]. It uses a feedback mechanism to securely distribute a single message to every correct process. The main challenge of Contagion is to ensure totality; we prove that the other properties are easily inherited from the underlying layer with high probability.

The basic idea of Contagion roughly corresponds to the last stage of Bracha's broadcast algorithm [12]. During the execution of Contagion for message m, processes first become *ready* for m. A correct process π can become ready for m in two ways:

1. π receives m from the underlying consistent broadcast layer.
2. π observes a certain fraction of other processes being ready for m.

A correct process delivers m only after it observes enough other processes being ready for m.

Unlike Bracha, we use samples (as opposed to quorums) to assess whether enough nodes are ready for m (and consequently our results are all probabilistic in nature). Upon initialization, every correct process selects a ready sample $\mathcal{R}$ of size R and a delivery sample $\mathcal{D}$ of size D. Our algorithm depends on four parameters: the *ready and delivery sample sizes* R and D, and the *ready and delivery thresholds* $\hat{R}$ and $\hat{D}$.

The delivery sample $\mathcal{D}$ is the sample used to assess whether m can be delivered. A correct process π delivers m if at least $\hat{D}$ out of the D processes in π's delivery sample are ready for m.

The purpose of the ready sample $\mathcal{R}$ is to create a feedback loop, a crucial part of the Contagion algorithm. When a correct process π observes at least $\hat{R}$ out of the R other processes in π's ready sample to be ready for m, π itself becomes ready for m. A direct consequence of such a feedback loop is the existence of a critical fraction of processes that, when ready for m, cause all the other correct processes become ready for m with high probability.

We require that $\hat{R}/R < \hat{D}/D$, i.e., the fraction of ready processes π needs to observe in order to become ready itself is smaller than the fraction of ready processed required for π to deliver m. Totality is then implied by the following intuitive argument. If a correct process π delivers m, it must have observed a fraction of at least $\hat{D}/D$ other processes being ready for m. As this fraction is higher than the critical fraction required for all correct processes to become ready for m, all correct processes will eventually become ready for m. Consequently, all correct processes will eventually deliver m. On the other hand, if too few processes are initially ready for m, such that the critical fraction is not reached, with high probability no correct process will observe the (even higher) fraction $\hat{D}/D$ of ready processes in its sample. Consequently, no correct process delivers m.

To broadcast a message m, the sender σ initially uses probabilistic consistent broadcast (Section 4) to disseminate m consistently to (a subset of) the correct processes. All correct processes that receive m through probabilistic consistent broadcast become ready for m. If their number is sufficiently high, according to the mechanism described above, all correct processes deliver m with high probability. If only a few correct processes deliver receive m from probabilistic consistent broadcast, with high probability no correct process delivers m.

5.2 Threshold Contagion Game

Before presenting the analysis of Contagion, we overview the *Threshold Contagion* game, an important tool in our analysis. In this game, we simulate the spreading of a contagious disease (without a cure) among members of a population, the same way the "readiness" for a message spreads among correct processes that execute our Contagion algorithm.

Threshold Contagion is played on the nodes of a directed multigraph, where each node represents a member of a population (whose state is either *infected* or *healthy*), and each edge represents a *can-infect* relation. An edge (a, b) means that a can infect b. We also call a the *predecessor* of b. In our Contagion algorithm, this corresponds to a being in the ready sample of b. Analogously to Contagion, a node becomes infected when enough of its predecessors are infected.

Threshold Contagion is played by one player in one or more *rounds*. At the beginning of each round, the player infects a subset of the healthy nodes. In the rest of the round, the infection (analogous to the readiness for a message) propagates as follows. A healthy

node that reaches a certain threshold ($\hat{R}$) of infected predecessors becomes infected as well (potentially contributing to the infection of more nodes). The round finishes when no healthy node has $\hat{R}$ or more infected predecessors, or when all nodes are infected.

In the analogy with our Contagion algorithm, infection by a player at the start of each round corresponds to a process receiving a message from the underlying probabilistic consistent broadcast layer. Infection through other nodes is analogous to observing $\hat{R}$ ready processes in the ready sample.

We analyze the Threshold Contagion game, and compute the probability distribution underlying the number of nodes that are infected at the end of a each round, depending on the number of healthy nodes infected by the player. Applying this analysis to the Contagion algorithm (the adversary being the player), we obtain the probability distribution of the number of processes ready for a message, which, in turn, allows us to compute a bound on the probability of violating the properties of Contagion. We provide all details on the Threshold Contagion game itself in [34, Appendix E].

5.3 Analysis Using Threshold Contagion

Here we present an outline of the analysis of Contagion; for a full formal treatment, see [34, Appendix C].

5.3.1 No duplication and integrity

Contagion deterministically satisfies these properties the same way as our previous algorithms do [34, Appendix C.3].

5.3.2 ϵ-Validity

Assuming a correct sender σ (by the premise of validity), we derive a bound on the probability ϵ of violating validity from the probability of the underlying probabilistic consistent broadcast failing and from the probability of σ's random delivery sample containig more than $D - \hat{D}$ Byzantine processes [34, Appendix C.4].

5.3.3 ϵ-Consistency

When computing the upper bound on the probability ϵ of compromising consistency [34, Appendix C.9], we assume that if the consistency of the underlying probabilistic consistent broadcast is compromised, then the consistency of probabilistic Byzantine reliable broadcast is compromised as well. The rest of the analysis assumes that probabilistic Byzantine reliable broadcast is consistent.

In such case, every correct process receives at most one message m^* from the underlying probabilistic consistent broadcast. Simply by acting correctly, Byzantine processes can cause any correct process to eventually deliver m^*. Consistency is compromised if the adversary can also cause at least one correct process to deliver a message $m \neq m^*$, given that no correct process becomes ready for m by receiving it through the underlying probabilistic consistent broadcast.

We start by noting that, since a correct process π can be ready for an arbitrary number of messages, the set of processes that are eventually ready for m is not affected by which processes are eventually ready for a message m^*. If enough processes in π's delivery sample are eventually ready both for m and m^*, then π can deliver either m or m^*. In this case, the adversary (who controls the network scheduling, see Section 2) decides which message π delivers.

The probability of m being delivered by any correct process is maximized when every Byzantine process behaves as if it was ready for m [34, Appendix C.9, Lemma 28]. Note that a Byzantine process being ready for m behaves identically to a correct process that receives m through probabilistic consistent broadcast. We model the adversarial system using a single-round game of Threshold Contagion where both correct and Byzantine processes are represented as nodes in the multigraph and all nodes representing Byzantine processes are initially infected [34, Appendix C.7, Lemma 26].

Given the distribution of the number of correct processes that are ready for m at the end Threshold Contagion, we compute the probability that at least one correct process will deliver $m \neq m^*$. This probability, combined with the probability that the consistency of probabilistic consistent broadcast is violated, yields the probability ϵ of violating the consistency of Contagion.

5.3.4 ϵ-Totality

Again, to compute an upper bound on the probability of our algorithm compromising totality, we assume that compromising the consistency of probabilistic consistent broadcast also compromises the totality of probabilistic Byzantine reliable broadcast. Assuming that probabilistic consistent broadcast satisfies consistency, at most one message m^* is received by any correct process through the underlying probabilistic consistent broadcast. We loosen the bound on the probability of compromising totality (and simplify analysis) by considering totality to be compromised if any message $m \neq m^*$ is delivered by any correct process. This allows us to focus on message m^*. We further loosen the bound by assuming that the Byzantine adversary can arbitrarily cause any correct process to become ready for m^*. Whenever this happens, zero or more additional correct processes will also become ready for m^* as a result of the feedback loop described in Section 5.1. To compromise totality, there must exists at least one correct process that delivers m^* and at least one correct process does not.

We prove [34, Appendix C.10.3, Lemma 31] that the optimal adversarial strategy to compromise totality is to repeat the following. (1) Make a correct node ready for m^*. (2) Wait until the "readiness" propagates to zero or more correct nodes. (3) Have specific Byzantine processes behave as correct processes ready for m^*, if this leads to some (but not all) correct processes delivering m^*. Totality is satisfied if, after every step of the adversary, either the feedback loop makes all correct processes deliver m^* (relying only on correct processes' ready samples), or no correct process delivers m^* (even with the "support" of Byzantine processes) [34, Theorem 14]. Otherwise, totality is violated.

We study this behavior with a multi-round game of Threshold Contagion, where only correct processes are represented as nodes in the multigraph and, at the beginning of each round, the player (i.e., the adversary) infects one uninfected node. From the probability distribution of the number of infected nodes after each round, we derive the probability of compromising totality by message m^*. This probability equals to the probability that there is at least one round after which the number of infected nodes allows some but not all the processes to deliver m^*.

6 Security and Complexity Evaluation

In Sections 3 to 5, we introduced three algorithms, Murmur, Sieve and Contagion, and outlined their analysis (deferring the formal details to the appendices).

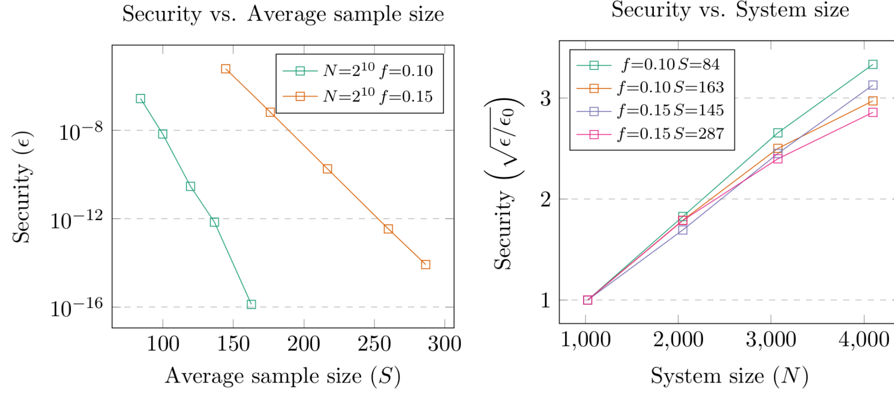


Figure 1 Left – ϵ-security of Contagion, as a function of the average sample size $S = \langle G, E, R, D\rangle$. We use a system size of 1024 processes and fractions of tolerated Byzantine processes $f = 0.1$ and $f = 0.15$. **Right** – Square root of the normalized ϵ-security of Contagion, as a function of the system size N, for various fractions of Byzantine processes (f) and average sample sizes (S). We normalize the values in each series by the first element of that series. All lines appearing to grow sub-linearly with a square-rooted y-axis demonstrates that the normalized ϵ security grows sub-quadratically.

The modular design of our algorithm allows us to study its components independently. We employ numerical techniques to maximize the ϵ-security of Contagion, under the constraint that the sum of all the sample sizes of a process is constant ($G + E + R + D = const$). Since a process communicates with all the processes in its samples, this corresponds to a fixed communication complexity.

For a given system size N and fraction of Byzantine processes f, we relate this per-process communication complexity to the ϵ-security of Contagion. As Figure 1 (left) shows, the probability ϵ of compromising the security of Contagion decays exponentially in the average sample size S.

We also study how the ϵ-security of Contagion changes as a function of the system size N, for a fixed set of parameters (G, E, R, D). Figure 1 (right) shows that the ϵ-security is bounded by a quadratic function in N. Thus, for a fixed security ϵ, the average sample size (and consequently, the communication complexity of our algorithm) grows logarithmically with the system size N.

Given that a process π only exchanges a constant number of messages with each member of π's samples, and the sample size is logarithmic in system size, each node needs to exchange $O(\log N)$ messages. Thus, for N nodes in the system, the overall message complexity is $O(N \log N)$. The latency in terms of message delays between broadcasting and delivery of a message is $O(\log N/ \log\log N)$. Specifically, the latency converges to $O(\log N/ \log\log N)$ message delays for gossip-based dissemination with Murmur (we prove this in [34, Appendix A.4, Theorem 5]), and 2 message delays in total for `Echo` (Sieve) and `Ready` (Contagion) messages.

7 Related Work

At its base, our broadcast algorithm relies on gossip. There is a great body of literature studying various aspects of gossip, proposing flavors of gossip protocols for different environments and analyzing their complexities [2, 6, 8, 7, 4, 20, 23, 30, 52, 28, 26, 27, 55, 57, 29, 36]. However, to the best of our knowledge, we propose the first highly scalable gossip-based reliable broadcast protocol resilient to Byzantine faults with a thorough probabilistic analysis.

The communication pattern in the implementation of both our Sieve and Contagion algorithms can be traced back to the Asynchronous Byzantine Agreement (ABA) primitive of Bracha and Toueg [13] and the subsequent line of work [12, 15, 42, 50]. Indeed, our echo-based mechanism in Sieve resembles algorithms from classic quorum-based systems for Byzantine consistent broadcast [53, 49]. The ready-based mechanism in Contagion is inspired by a two-phase protocol appearing in several practical (quorum-based) systems [15, 19, 44]. Compared to classic work on this topic, the key feature of Contagion and Sieve is that they replace the building block of quorum systems with stochastic samples, thus enabling better scalability for the price of abandoning deterministic guarantees.

There is significant prior work on using epidemic algorithms to implement scalable *reliable* broadcast [9, 22, 37, 41]. Under benign failures or constant churn, these algorithms ensure, with high probability, that every broadcast message reaches all or none, and that all messages from correct senders are delivered. Our goal is to additionally provide *consistency* for broadcast messages, and tolerate *Byzantine* environments [13, 45, 53]. To the best of our knowledge, we are the first to apply the epidemic sample-based methodology in this context. Our main algorithm Contagion scales well to dynamic systems of thousands of nodes, some of which may be Byzantine. This makes it a suitable choice for *permissionless* settings that are gaining popularity with the advent of blockchains [47].

Distributed clustering techniques seek to group the processes of a system into clusters, sometimes called shards or quorums, of size $O(logN)$ [5, 31, 32, 38, 39, 51]. This line of work has various goals (e.g., leader election, "almost everywhere" agreement, building an overlay network) and they also aim for scalable solutions. The overarching principle in clustering techniques is similar to our use of samples: build each cluster in a provably random manner so that the adversary cannot dominate any single cluster. Samples in our solution are private and individual on a per-process basis, in contrast to clusters which are typically public and global for the whole system.

The idea of *communication locality* appears in the context of secure multi-party computation (MPC) protocols [11, 16, 24]. This property captures the intuition that, in order to obtain scalable distributed protocols and permit a large number of participants, it is desirable to limit the number of participants each process must communicate with. All of our three algorithms have this communication locality property, since each process coordinates only with logarithmically-sized samples. In contrast to secure MPC protocols, our algorithms have different goals, system model, or assumptions (e.g., we do not assume a client-server model [24], nor do we seek to address privacy issues). Our algorithms can be used as building blocks towards helping tackle scalability in MPC protocols, and we consider this an interesting avenue for future work.

References

1. Daron Acemoglu and Asu Ozdaglar. 6.207/14.15: Networks - Lecture 4: Erdős–Rényi Graphs and Phase Transitions, 2009. URL: `https://economics.mit.edu/files/4622`.
2. Dan Alistarh, Seth Gilbert, Rachid Guerraoui, and Morteza Zadimoghaddam. How Efficient Can Gossip Be? (on the Cost of Resilient Information Exchange). In *Proceedings of the 37th International Colloquium Conference on Automata, Languages and Programming: Part II*, ICALP'10, pages 115–126, Berlin, Heidelberg, 2010. Springer-Verlag. URL: `http://dl.acm.org/citation.cfm?id=1880999.1881012`.
3. Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *JACM*, 42(1), 1995.
4. Chen Avin, Michael Borokhovich, Keren Censor-Hillel, and Zvi Lotker. Order Optimal Information Spreading Using Algebraic Gossip. In *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '11, pages 363–372, New York, NY, USA, 2011. ACM. `doi:10.1145/1993806.1993883`.

5 Baruch Awerbuch and Christian Scheideler. Towards a scalable and robust DHT. *Theory of Computing Systems*, 45(2):234–260, 2009.

6 Petra Berenbrink, Robert Elsaesser, and Tom Friedetzky. Efficient Randomised Broadcasting in Random Regular Networks with Applications in Peer-to-peer Systems. In *Proceedings of the Twenty-seventh ACM Symposium on Principles of Distributed Computing*, PODC '08, pages 155–164, New York, NY, USA, 2008. ACM. doi:10.1145/1400751.1400773.

7 Petra Berenbrink, Robert Elsässer, and Thomas Sauerwald. Communication Complexity of Quasirandom Rumor Spreading. In *Proceedings of the 18th Annual European Conference on Algorithms: Part I*, ESA'10, pages 134–145, Berlin, Heidelberg, 2010. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=1888935.1888952.

8 Petra Berenbrink, Robert Elsässer, and Thomas Sauerwald. Randomised Broadcasting: Memory vs. Randomness. *Theoretical Computer Science*, 520:306–319, April 2010. doi:10.1007/978-3-642-12200-2_28.

9 Kenneth P. Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. Bimodal Multicast. *ACM Trans. Comput. Syst.*, 17(2):41–88, May 1999. doi:10.1145/312203.312207.

10 Edward Bortnikov, Maxim Gurevich, Idit Keidar, Gabriel Kliot, and Alexander Shraer. Brahms: Byzantine resilient random membership sampling. *Computer Networks*, 53(13):2340–2359, 2009. Gossiping in Distributed Systems. doi:10.1016/j.comnet.2009.03.008.

11 Elette Boyle, Shafi Goldwasser, and Stefano Tessaro. Communication Locality in Secure Multi-party Computation. In *Theory of Cryptography*, 2013.

12 Gabriel Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.

13 Gabriel Bracha and Sam Toueg. Asynchronous Consensus and Broadcast Protocols. *JACM*, 32(4), 1985.

14 Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming.* Springer Publishing Company, Incorporated, 2nd edition, 2011.

15 Christian Cachin and Jonathan A. Poritz. Secure Intrusion-tolerant Replication on the Internet. In *DSN*, 2002.

16 Nishanth Chandran, Wutichai Chongchitmate, Juan A. Garay, Shafi Goldwasser, Rafail Ostrovsky, and Vassilis Zikas. The Hidden Graph Model: Communication Locality and Optimal Resiliency with Adaptive Faults. In *ITCS '15*, 2015.

17 Fan Chung and Linyuan Lu. The Diameter of Sparse Random Graphs. *Advances in Applied Mathematics*, 26:257–279, 2001.

18 Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The Second-generation Onion Router. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 21–21, Berkeley, CA, USA, 2004. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=1251375.1251396.

19 Sisi Duan, Michael K. Reiter, and Haibin Zhang. BEAT: Asynchronous BFT Made Practical. In *CCS*, 2018.

20 Robert Elsässer and Dominik Kaaser. On the Influence of Graph Density on Randomized Gossiping. *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 521–531, 2015.

21 Paul Erdös and Alfréd Rényi. On Random Graphs. *Publicationes Mathematicae*, 6:290–297, 1959.

22 P. Th. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A.-M. Kermarrec. Lightweight Probabilistic Broadcast. *ACM Trans. Comput. Syst.*, 21(4):341–374, November 2003. doi:10.1145/945506.945507.

23 Yaacov Fernandess, Antonio Fernández, and Maxime Monod. A Generic Theoretical Framework for Modeling Gossip-based Algorithms. *SIGOPS Oper. Syst. Rev.*, 41(5):19–27, October 2007. doi:10.1145/1317379.1317384.

24 Juan Garay, Yuval Ishai, Rafail Ostrovsky, and Vassilis Zikas. The price of low communication in secure multi-party computation. In *Annual International Cryptology Conference*, pages 420–446. Springer, 2017.

25 Juan A Garay, Jonathan Katz, Ranjit Kumaresan, and Hong-Sheng Zhou. Adaptively Secure Broadcast, Revisited. In *PODC*, pages 179–186, 2011.

26 Chryssis Georgiou, Seth Gilbert, Rachid Guerraoui, and Dariusz R. Kowalski. On the Complexity of Asynchronous Gossip. In *Proceedings of the Twenty-seventh ACM Symposium on Principles of Distributed Computing*, PODC '08, pages 135–144, New York, NY, USA, 2008. ACM. doi:10.1145/1400751.1400771.

27 Chryssis Georgiou, Seth Gilbert, Rachid Guerraoui, and Dariusz R. Kowalski. Asynchronous Gossip. *J. ACM*, 60(2):11:1–11:42, May 2013. doi:10.1145/2450142.2450147.

28 Chryssis Georgiou, Seth Gilbert, and Dariusz R. Kowalski. Meeting the deadline: on the complexity of fault-tolerant continuous gossip. *Distributed Computing*, 24(5):223–244, December 2011. doi:10.1007/s00446-011-0144-6.

29 Mohsen Ghaffari and Merav Parter. A Polylogarithmic Gossip Algorithm for Plurality Consensus. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC '16, pages 117–126, New York, NY, USA, 2016. ACM. doi:10.1145/2933057.2933097.

30 George Giakkoupis, Yasamin Nazari, and Philipp Woelfel. How Asynchrony Affects Rumor Spreading Time. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC '16, pages 185–194, New York, NY, USA, 2016. ACM. doi:10.1145/2933057.2933117.

31 Rachid Guerraoui, Florian Huc, and Anne-Marie Kermarrec. Highly dynamic distributed computing with byzantine failures. In *PODC*, 2013. URL: http://dl.acm.org/citation.cfm?doid=2484239.2484263.

32 Rachid Guerraoui, Anne-Marie Kermarrec, Matej Pavlovic, and Dragos-Adrian Seredinschi. Atum: Scalable Group Communication Using Volatile Groups. In *Proceedings of the 17th International Middleware Conference*, Middleware '16, pages 19:1–19:14, New York, NY, USA, 2016. ACM. doi:10.1145/2988336.2988356.

33 Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos Seredinschi. The Consensus Number of a Cryptocurrency. In *PODC*, 2019.

34 Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. Scalable Byzantine Reliable Broadcast (Extended Version). *arXiv preprint arXiv:1908.01738, Version 1*, 2019. arXiv:1908.01738v1.

35 Vassos Hadzilacos and Sam Toueg. Fault-tolerant broadcasts and related problems. In Sape J. Mullender, editor, *Distributed Systems*, chapter 5, pages 97–145. Addison-Wesley, 1993.

36 Bernhard Haeupler, Gopal Pandurangan, David Peleg, Rajmohan Rajaraman, and Zhifeng Sun. Discovery Through Gossip. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 140–149, New York, NY, USA, 2012. ACM. doi:10.1145/2312005.2312031.

37 Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. T-Man: Gossip-based Fast Overlay Topology Construction. *Comput. Netw.*, 53(13):2321–2339, August 2009. doi:10.1016/j.comnet.2009.03.013.

38 Valerie King, Steven Lonargan, Jared Saia, and Amitabh Trehan. Load Balanced Scalable Byzantine Agreement through Quorum Building, with Full Information. In *International Conference on Distributed Computing and Networking*, pages 203–214. Springer, 2011.

39 Valerie King, Jared Saia, Vishal Sanwalani, and Erik Vee. Scalable Leader Election. In *SODA*, 2006.

40 Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *TOPLAS*, 4(3), 1982.

41 Meng-Jang Lin, Keith Marzullo, and Stefano Masini. Gossip Versus Deterministically Constrained Flooding on Small Networks. In *Proceedings of the 14th International Conference on Distributed Computing*, DISC '00, pages 253–267, London, UK, UK, 2000. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=645957.675970.

42 Dahlia Malkhi, Michael Merritt, and Ohad Rodeh. Secure Reliable Multicast Protocols in a WAN. In *ICDCS*, 1997.

43 Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 569–578. ACM, 1997.

44 Dahlia Malkhi and Michael K. Reiter. A High-Throughput Secure Reliable Multicast Protocol. In *CSFW*, 1996.

45 Dahlia Malkhi and Michael K. Reiter. A High-Throughput Secure Reliable Multicast Protocol. *Journal of Computer Security*, 5(2):113–128, 1997. `doi:10.3233/JCS-1997-5203`.

46 Dahlia Malkhi, Michael K Reiter, Avishai Wool, and Rebecca N Wright. Probabilistic Quorum Systems. *Inf. Comput.*, 170(2):184–206, November 2001. `doi:10.1006/inco.2001.3054`.

47 Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008.

48 Fernando Pedone and André Schiper. Handling message semantics with generic broadcast protocols. *Distributed Computing*, 15(2):97–107, 2002.

49 Michael K. Reiter. Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart. In *CCS*, 1994.

50 Michael K. Reiter and Kenneth P. Birman. How to securely replicate services. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3), 1994.

51 Christian Scheideler. How to Spread Adversarial Nodes? Rotate! In *STOC*, pages 704–713. ACM, 2005.

52 Suman Sourav, Peter Robinson, and Seth Gilbert. Slow Links, Fast Links, and the Cost of Gossip. *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 786–796, 2018.

53 Sam Toueg. Randomized Byzantine Agreements. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, PODC '84, pages 163–178, New York, NY, USA, 1984. ACM. `doi:10.1145/800222.806744`.

54 Jelle van den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: Scalable Private Messaging Resistant to Traffic Analysis. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 137–152, New York, NY, USA, 2015. ACM. `doi:10.1145/2815400.2815417`.

55 Spyros Voulgaris, Márk Jelasity, and Maarten van Steen. A Robust and Scalable Peer-to-peer Gossiping Protocol. In *Proceedings of the Second International Conference on Agents and Peer-to-Peer Computing*, AP2PC'03, pages 47–58, Berlin, Heidelberg, 2004. Springer-Verlag. `doi:10.1007/978-3-540-25840-7_6`.

56 Marko Vukolic. The Origin of Quorum Systems. *Bulletin of the EATCS*, 101:125–147, 2010. URL: `http://eatcs.org/beatcs/index.php/beatcs/article/view/183`.

57 B. Zhang, K. Han, B. Ravindran, and E. D. Jensen. RTQG: Real-Time Quorum-based Gossip Protocol for Unreliable Networks. In *2008 Third International Conference on Availability, Reliability and Security*, pages 564–571, March 2008. `doi:10.1109/ARES.2008.139`.

Fast Distributed Algorithms for LP-Type Problems of Low Dimension

Kristian Hinnenthal
Paderborn University, Germany
krijan@mail.upb.de

Christian Scheideler
Paderborn University, Germany
scheideler@upb.de

Martijn Struijs
TU Eindhoven, The Netherlands
m.a.c.struijs@tue.nl

Abstract

In this paper we present various distributed algorithms for LP-type problems in the well-known gossip model. LP-type problems include many important classes of problems such as (integer) linear programming, geometric problems like smallest enclosing ball and polytope distance, and set problems like hitting set and set cover. In the gossip model, a node can only push information to or pull information from nodes chosen uniformly at random. Protocols for the gossip model are usually very practical due to their fast convergence, their simplicity, and their stability under stress and disruptions. Our algorithms are very efficient (logarithmic rounds or better with just polylogarithmic communication work per node per round) whenever the combinatorial dimension of the given LP-type problem is constant, even if the size of the given LP-type problem is polynomially large in the number of nodes.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Theory of computation → Mathematical optimization

Keywords and phrases LP-type problems, linear optimization, distributed algorithms, gossip algorithms

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.23

Funding K. Hinnenthal and C. Scheideler are supported by the DFG SFB 901 (On-The-Fly Computing).

1 Introduction

1.1 LP-type Problems

LP-type problems were defined by Sharir and Welzl [18] as problems characterized by a tuple (H, f) where H is a finite set of constraints and $f : 2^H \to T$ is a function that maps subsets from H to values in a totally ordered set $(T, \leq)$ containing $-\infty$. The function f is required to satisfy two conditions:

- **Monotonicity:** For all sets $F \subseteq G \subseteq H$, $f(F) \leq f(G) \leq f(H)$.
- **Locality:** For all sets $F \subseteq G \subseteq H$ with $f(F) = f(G)$ and every element $h \in H$, if $f(G) < f(G \cup \{h\})$ then $f(F) < f(F \cup \{h\})$.

Given an LP-type problem (H, f), the goal is to determine $f(H)$. In doing so, the following notation has commonly been used. A subset $B \subseteq H$ with $f(B') < f(B)$ for all proper subsets B' of B is called a *basis* of H. An *optimal basis* is a basis B with $f(B) = f(H)$. The maximum cardinality of a basis is called the *(combinatorial) dimension* of (H, f) and denoted by $\dim(H, f)$.

© Kristian Hinnenthal, Christian Scheideler, and Martijn Struijs;
licensed under Creative Commons License CC-BY
33rd International Symposium on Distributed Computing (DISC 2019).
Editor: Jukka Suomela; Article No. 23; pp. 23:1–23:16

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

LP-type problems cover many important optimization problems like linear optimization problems (where H is the set of linear inequalities, $f(G)$ represents the optimal solution w.r.t. the given objective function under the constraints $G \subseteq H$, and the dimension is at most the number of variables in the LP) or geometric problems like the smallest enclosing ball problem (where H is the set of points, $f(G)$ is the radius of the smallest enclosing ball for point set $G \subseteq H$, and the dimension is at most $d+1$ for the d-dimensional case).

Clarkson [2] proposed a very elegant randomized algorithm for solving LP-type problems (see Algorithm 1). In this algorithm, each $h \in H$ has a multiplicity of $\mu_h \in \mathbb{N}$, and $H(\mu)$ is a multiset where each $h \in H$ occurs μ_h times in $H(\mu)$. The algorithm requires a subroutine for computing $f(S)$ for sets S of size $O(\dim(H,f)^2)$, but this is usually straightforward if $\dim(H,f)$ is a constant. In the following, let $d = \dim(H,f)$, and we say that an iteration of the repeat-loop is *successful* if $|V| \leq |H(\mu)|/(3d)$, where V is the set of elements violating a solution $f(R)$ for the chosen subset $R \subseteq H(\mu)$ (which might be a multiset), see also the algorithm.

Algorithm 1 Clarkson's Algorithm.

```
1:  if |H| ≤ 6 dim(H, f)^2 then return f(H)
2:  else
3:      r := 6 dim(H, f)^2
4:      for all h ∈ H do μ_h := 1
5:      repeat
6:          choose a random subset R of size r from H(μ)
7:          V := {h ∈ H(μ) | f(R) < f(R ∪ {h})}
8:          if |V| ≤ |H(μ)|/(3 dim(H, f)) then
9:              for all h ∈ V do μ_h := 2μ_h
10:     until V = ∅
11:     return f(R)
```

▶ **Lemma 1** ([9]). *Let (H,f) be an LP-type problem of dimension d and let μ be any multiplicity function. For any $1 \leq r < m$, where $m = |H(\mu)|$, the expected size of $V = \{h \in H(\mu) \mid f(R) < f(R \cup \{h\})\}$ for a random subset R of size r from $H(\mu)$ is at most $d \cdot \frac{m-r}{r+1}$.*

From this lemma and the Markov inequality it immediately follows that the probability that an iteration of the repeat-loop is successful is at least 1/2. Moreover, it holds:

▶ **Lemma 2** ([15, 20]). *Let $k \in \mathbb{N}$ and B be an arbitrary optimal basis of H. After $k \cdot d$ successful iterations, $2^k \leq \mu(B) < |H| \cdot e^{k/3}$.*

Lemma 2 implies that Clarkson's algorithm must terminate after at most $O(d \log |H|)$ successful iterations (as otherwise $2^k > |H| \cdot e^{k/3}$), so Clarkson's algorithm performs at most $O(d \log |H|)$ iterations of the repeat-loop, on expectation. This bound is also best possible in the worst case for any $d \ll |H|$: given that there is a unique optimal basis B of size d, its elements can have a multiplicity of at most $\sqrt{|H|}$ after $(\log |H|)/2$ iterations, so the probability that B is contained in R is polynomially small in $|H|$ up to that point.

Clarkson's algorithm can easily be transformed into a distributed algorithm with expected runtime $O(d \log^2 n)$ if n nodes are available that are interconnected by a hypercube, $n = |H|$, and each node is responsible for one element in H, for example, because in that case every

iteration of the repeat-loop can be emulated in $O(\log n)$ communication rounds, w.h.p.[1], using appropriate (weighted random) routing, broadcast, and convergecast approaches. However, it has been open so far whether it is also possible to construct a distributed algorithm for LP-type problems with an expected runtime of just $O(d \log n)$ (either with a variant of Clarkson's algorithm or a different approach). We will show in this paper that this is possible when running certain variants of Clarkson's algorithm in the gossip model, even if H has a size polynomial in n.

1.2 Network Model

We assume that we are given a fixed node set U (instead of the standard notation V to distinguish it from the violator set V) of size n consisting of the nodes $v_1, \ldots, v_n$. In our paper, we do not require the nodes to have IDs. Moreover, we assume the standard synchronous message passing model, i.e., the nodes operate in synchronous *(communication) rounds*, and all messages sent (or requested) in round i will be received at the beginning of round $i+1$.

In the (uniform) gossip model, a node can only send or receive messages via random *push* and *pull* operations. In a push operation, it can send a message to a node chosen uniformly at random while in a pull operation, it can ask a node chosen uniformly at random to send it a message. We will restrict the message size (i.e., its number of bits) to $O(\log n)$. A node may execute multiple push and pull operations in parallel in a round. The number of push and pull operations executed by it in a single round is called its *(communication) work*.

Protocols for the gossip model are usually very practical due to their fast convergence, their simplicity, and their stability under stress and disruptions. Many gossip-based protocols have already been presented in the past, including protocols for information dissemination, network coding, load-balancing, consensus, and quantile computations (see [3, 11, 12, 13, 14] for some examples). Also, gossip protocols can be used efficiently in the context of population protocols and overlay networks, two important areas of network algorithms. In fact, it is easy to see that any algorithm with runtime T and maximum work W in the gossip model can be emulated by overlay networks in $O(T + \log n)$ time and with maximum work $O(W \log n)$ w.h.p. (since it is easy to set up n (near-)random overlay edges, one per node, in hypercubic networks in $O(\log n)$ time and with $O(\log n)$ work, w.h.p., and this can be pipelined to avoid a $\log n$-overhead in the runtime).

1.3 Related Work

There has already been a significant amount of work on finding efficient sequential and parallel algorithms for linear programs of low dimension (i.e., based on a small number of variables), which are a special case of LP-type problems of low combinatorial dimension (see [5] for a very thorough survey). We just focus here on parallel algorithms. The fastest parallel algorithm known for the CRCW PRAM is due to Alon and Megiddo [1], which has a runtime of $O(d^2 \log^2 d)$. It essentially follows the idea of Clarkson, with the main difference that it replicates elements in V much more aggressively by exploiting the power of the CRCW PRAM. This is achieved by first compressing the violated elements into a small area and then replicating them by a factor of $n^{1/(4d)}$ (instead of just 2). The best work-optimal algorithm for the CRCW PRAM is due to Goodrich [10], which is based on an algorithm by Dyer and Frieze [6] and has a runtime of $O((\log \log n)^d)$. This also implies a work-optimal algorithm

[1] By "with high probability", or short, "w.h.p.", we mean a probability of least $1 - 1/n^c$ for any constant $c > 0$.

for the EREW PRAM, but the runtime increases to $O(\log n(\log\log n)^d)$ in this case. The fastest parallel algorithm known for the EREW PRAM is due to Dyer [4], which achieves a runtime of $O(\log n(\log\log n)^{d-1})$ when using an $O(\log n)$-time parallel sorting algorithm (like Cole's algorithm). Since the runtime of any algorithm for solving a linear program of constant dimension in an EREW PRAM is known to be $\Omega(\log n)$ [5], the upper bound is optimal for $d = 1$.

Due to Ranade's seminal work [16], it is known that any CRCW PRAM step can be emulated in a butterfly network in $O(\log n)$ communication rounds, yielding an $O(d^2 \log^2 d \log n)$-time algorithm for linear programs of constant dimension in the butterfly. However, it is not clear whether any of the parallel algorithms would work for arbitrary LP-type problems. Also, none of the proposed parallel algorithms seem to be easily adaptable to an algorithm that works efficiently (i.e., in time $o(\log^2 n)$ and with *polylog* work) for the gossip model as they require processors to work together in certain groups or on certain memory locations in a coordinated manner, and assuming no (unique) node IDs would further complicate the matter.

As mentioned above, LP-type problems were introduced by Sharir and Welzl [18]. Since then, various results have been shown, but only for sequential algorithms. Combining results by Gärtner [7] with Clarkson's methods, Gärtner and Welzl [8] proposed an algorithm with runtime $O(d^2|H|) + e^{O(\sqrt{d \log d})}$, for any d, which implies that as long as $d = O(\log^2 |H|/\log\log|H|)$, $f(H)$ can be determined in polynomial time. Extensions of LP-type problems were studied by Gärtner [7] (abstract optimization problems) and Skovron [19] (violator spaces).

1.4 Our Results

In all of our results, we assume that initially, H is randomly distributed among the nodes. This is easy to achieve in the gossip model if this is not the case (for example, each node initially represents its own point for the smallest enclosing ball problem) by performing a push operation on each element. The nodes are assumed to know f, and we require the nodes to have a constant factor estimate of $\log n$ for the algorithms to provide a correct output, w.h.p., but they may not have any information about $|H|$. For simplicity, we also assume that the nodes know d. If not, they may perform a binary search on d (by stopping the algorithm and switching to $2d$ if it takes too long for some d), which does not affect our bounds below since they depend at least linearly on d.

In all of our results and proofs we assume that the dimension d of the given LP-type problem is at most logarithmic in $|H|$, to ensure that internal computations only take polynomial time and message sizes satisfy our $O(\log n)$ bound. Section 2 starts with the lightly loaded case (i.e., $|H| = O(n \log n)$, where $n = |U|$) and proves the following theorem.

▶ **Theorem 3.** *For any LP-type problem (H, f) satisfying $|H| = O(n \log n)$, the Low-Load Clarkson Algorithm finds an optimal solution in $O(d \log n)$ rounds with maximum work $O(d^2 + \log n)$ per round, w.h.p.*

At a high level, the Low-Load Clarkson Algorithm is similar to the original Clarkson algorithm, but sampling a random subset and termination detection are more complex now, and a filtering approach is needed to keep $|H(\mu)|$ low at all times so that the work is low. In Section 3, we then consider the highly loaded case and prove the following theorem.

▶ **Theorem 4.** *For any LP-type problem (H, f) with $|H| = \omega(n \log n)$ and $|H| = O(poly(n))$, the High-Load Clarkson Algorithm finds an optimal solution in $O(d \log n)$ rounds with maximum work $O(d \log n)$ per round, w.h.p. If we allow a maximum work of $O(d \log^{1+\epsilon} n)$ per round, for any constant $\epsilon > 0$, the runtime reduces to $O(d \log(n)/\log\log(n))$, w.h.p.*

Note that as long as we only allow the nodes to spend polylogarithmic work per round, a trivial lower bound on the runtime when using Clarkson's approach is $\Omega(\log(n)/\log\log(n))$ since in $o(\log(n)/\log\log(n))$ rounds an element in H can only be spread to $n^{o(1)}$ nodes, so the probability of fetching it under the gossip model is minute.

The reason why we designed different algorithms for the lightly loaded and highly loaded cases is that the Low-Load Clarkson Algorithm is much more efficient than the High-Load Clarkson Algorithm concerning internal computations. Also, it is better concerning the work for the lightly loaded case, but its work does not scale well with an increasing $|H|$. The main innovation for Theorem 4 is that we come up with a Chernoff-style bound for $|V|$ that holds for all LP-type problems. Gärtner and Welzl [9] also provided a Chernoff-style bound on $|V|$ for LP-type problems, but their proof only works for LP-type problems that are regular (i.e., for all $G \subseteq H$ with $|G| \geq d$, all optimal bases of G have a size of exactly d) and non-degenerate (i.e., every $G \subset H$ with $|G| \geq d$ has a unique optimal basis). While regularity can be enforced in the non-degenerate case, it is not known so far how to make a general LP-type problem non-degenerate without substantially changing its structure (though for most of the applications considered so far for LP-type problems, slight perturbations of the input would solve this problem). Also, since the duplication approach of Clarkson's algorithm generates degenerate instances, their Chernoff-style bound therefore cannot be used here.

2 Low-Load Clarkson Algorithm

Suppose that we have an arbitrary LP-type problem (H, f) of dimension d with $|H| = O(n \log n)$. First, we present and analyze an algorithm for $|H| \geq n$, and then we extend it to any $1 \leq |H| = O(n \log n)$.

Recall that initially the elements of H are assigned to the nodes uniformly and independently at random. Let us denote the set of these elements in node v_i by $H_0(v_i)$ to distinguish them from copies created later by the algorithm, and let $H_0 = \bigcup_i H_0(v_i)$.

At any time, $H(v_i)$ denotes the (multi)set of elements in H known to v_i (including the elements in H_0) and $H(U) = \bigcup_i H(v_i)$, where U represents the node set. Let $m = |H(U)|$. At a high level, our distributed algorithm is similar to the original Clarkson algorithm, but sampling a random subset and termination detection are more complex now (which will be explained in dedicated subsections). In fact, the sampling might fail since a node v_i might not be able to collect enough elements for its sample set R_i. Also, a filtering approach is needed to keep $|H(U)|$ low at all times (see Algorithm 2). However, it will never become too low since the algorithm never deletes an element in H_0, so $|H(U)| \geq n$ at any time. Note that never deleting an element in H_0 also guarantees that no element in H will ever be washed out (which would result in incorrect solutions).

For the runtime analysis, we note that sampling R_i can be done in one round (see Section 2.1), spreading the violator set V_i just takes one round (by executing the push operations in parallel), and we just need one more round for processing the received elements h, so each iteration of the repeat loop just takes $O(1)$ rounds. We start with a slight variant of Lemma 1.

▶ **Lemma 5.** *Let (H, f) be an LP-type problem of dimension d. For any $1 \leq r < m$, where $m = |H(U)|$, the expected size of $V_i = \{h \in H(v_i) \mid f(R) < f(R \cup \{h\})\}$ for a random subset R of size r from $H(U)$ is at most $d \cdot \frac{m-r}{n(r+1)}$.*

Proof. According to Lemma 1, the expected size of $V(R) = \{h \in H(V) \mid f(R) < f(R \cup \{h\})\}$ for a random subset R is at most $d \cdot \frac{m-r}{r+1}$. Since every element in $H(U)$ has a probability of $1/n$ to belong to $H(v_i)$, $\mathbb{E}[|V_i|] \leq d \cdot \frac{m-r}{n(r+1)}$. ◀

Algorithm 2 Low-Load Clarkson Algorithm.

```
1:  repeat
2:      for all nodes v_i in parallel do
3:          choose a random subset R_i of size 6d^2 from H(U)        ▷ see Section 2.1
4:          if the sampling of R_i succeeds then
5:              V_i := {h ∈ H(v_i) | f(R_i) < f(R_i ∪ {h})}
6:              for all h ∈ V_i do push(h)                          ▷ randomly spread V_i
7:          for all h received by v_i do add h to H(v_i)
8:          for all h ∈ H(v_i) − H_0(v_i) do
9:              keep h with probability 1/(1 + 1/(2d))
10: until at least one v_i satisfies f(R_i) = f(H)                    ▷ see Section 2.2
```

This allows us to prove the following lemma.

▶ **Lemma 6.** *For all i, $|V_i| = O(m/n + \log n)$, w.h.p., and $\sum_{i=1}^{n} |V_i| \leq m/(3d)$, w.h.p.*

Proof. Let the random variable X_i be defined as $|V_i|$ and let $X = \sum_i X_i$. If the sampling of R_i fails then, certainly, $X_i = 0$, and otherwise, $\mathbb{E}[X_i] \leq d \cdot \frac{m-r}{n(r+1)}$ for all i. Thus, $\mathbb{E}[X] \leq d \cdot \frac{m-r}{r+1}$. Also, since the elements in $H(U)$ are distributed uniformly and independently at random among the nodes at all times, the standard Chernoff bounds imply that $|H(v_i)| = O(m/n + \log n)$ w.h.p., and therefore also $X_i \leq O(m/n + \log n)$ w.h.p. Unfortunately, the X_i's are not independent since the $H(v_i)$'s are not chosen independently of each other though the R_i's are, but the dependencies are minute: given that we have already determined $H(v_j)$ for k many v_j's, where $k = o(n)$ is sufficiently small, the probability that any one of the remaining elements $h \in H(U)$ is assigned to $H(v_i)$ is $1/(n - o(n)) = (1 + o(1))/n$, so that for any subset $S = \{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\}$ of size k,

$$\begin{aligned}
\mathbb{E}[X_{i_1} \cdot \ldots \cdot X_{i_k}] &= \sum_{x_{i_1},\ldots,x_{i_k}} x_{i_1} \cdot \ldots \cdot x_{i_k} \cdot \Pr[X_{i_1} = x_{i_1} \;\wedge \ldots \wedge\; X_{i_k} = x_{i_k}] \\
&= \sum_{H(v_{i_1})} \Pr[H(v_{i_1})] \sum_{x_{i_1}} x_{i_1} \Pr[X_{i_1} = x_{i_1} \mid H(v_{i_1})] \cdot \\
&\quad \left(\sum_{H(v_{i_2})} \Pr[H(v_{i_2}) \mid H(v_{i_1})] \sum_{x_{i_2}} x_{i_2} \Pr[X_{i_2} = x_{i_2} \mid H(v_{i_1}) \wedge H(v_{i_2})] \cdot \ldots \right) \\
&= \sum_{H(v_{i_1})} \Pr[H(v_{i_1})] \sum_{x_{i_1}} x_{i_1} \Pr[X_{i_1} = x_{i_1} \mid H(v_{i_1})] \cdot \\
&\quad \left(\sum_{H(v_{i_2})} (1 + o(1)) \Pr[H(v_{i_2})] \sum_{x_{i_2}} x_{i_2} \Pr[X_{i_2} = x_{i_2} \mid H(v_{i_2})] \cdot \ldots \right) \\
&= \ldots \leq (1 + o(1))^k \prod_{j=1}^{k} \mathbb{E}[X_{i_j}] \leq \left((1 + o(1))\, d \cdot \frac{m-r}{n(r+1)} \right)^k . \qquad (*)
\end{aligned}$$

This allows us to use a Chernoff-Hoeffding-style bound for k-wise negatively correlated random variables, which is a slight extension of Theorem 3 in [17]:

▶ **Theorem 7.** *Let $X_1, \dots, X_n$ be random variables with $X_i \in [0, C]$ for some $C > 0$. Suppose there is a $k > 1$ and $q > 0$ with $\mathbb{E}[\prod_{i \in S} X_i] \leq q^s$ for all subsets $S \subseteq \{1, \dots, n\}$ of size $s \leq k$. Let $X = \sum_{i=1}^n X_i$ and $\mu = q \cdot n$. Then it holds for all $\delta > 0$ with $k \geq \lceil \mu\delta \rceil$ that*

$$\Pr[X \geq (1+\delta)\mu] \leq e^{-\min\{\delta^2, \delta\}\mu/(3C)}$$

Setting $C = \Theta(m/n + \log n)$, $\mu = q \cdot n$ with $q = (1+o(1))d \cdot \frac{m-r}{n(r+1)}$ and $r = 6d^2$, and $\delta > 0$ large enough so that $\delta^2\mu = \omega(C \ln n)$ but $\delta\mu = o(n)$ so that inequality $(*)$ applies, which works for $\delta = \Theta(\sqrt{(Cd \ln n)/m}) = O(\sqrt{(d \ln^2 n)/n})$, $\Pr[X \geq m/(3d)]$ is polynomially small in n. ◀

Next, we show that m will never be too large, so that the communication work of the nodes will never be too large.

▶ **Lemma 8.** *For up to polynomially many iterations of the Low-Load Clarkson Algorithm, $|H(U)| = O(|H_0|)$, w.h.p.*

Proof. Let $q = |H(U) - H_0|$ and suppose that $|H(U)| \geq c|H_0|$ for some $c \geq 4$, which implies that $q \geq (c-1)/c \cdot m$. Then it holds for the size q' of $H(U) - H_0$ at the end of a repeat-round that

$$\begin{aligned}
\mathbb{E}[q'] &\leq \left(q + \frac{m}{3d}\right) \cdot 1/\left(1 + \frac{1}{2d}\right) \leq \left(q + \frac{cq}{(c-1) \cdot 3d}\right) / \left(1 + \frac{1}{2d}\right) \\
&= q\left(\left(1 + \frac{1}{2d}\right) - \left(\frac{1}{6d} - \frac{1}{(c-1) \cdot 3d}\right)\right) / \left(1 + \frac{1}{2d}\right) \\
&= q(1 - \Theta(1/d))
\end{aligned}$$

Since the decision to keep elements $h \in H(U) - H_0$ is done independently for each h, it follows from the Chernoff bounds that $\Pr[q' > q]$ is polynomially small in n for $|H(U)| \geq 4|H_0|$. Moreover, Lemma 6 implies that $|H(U)|$ can increase by at most $|H(U)|/3$ in each iteration, w.h.p., so for polynomially many iterations of the algorithm, $|H(U)| \leq 5|H_0|$ w.h.p. ◀

Thus, combining Lemma 6 and Lemma 8, the maximum work per round for pushing out some V_i is bounded by $O(\log n)$ w.h.p. Next we prove a lemma that adapts Lemma 2 to our setting.

▶ **Lemma 9.** *Let B be an arbitrary optimal basis of H. If, for T many iterations of the Low-Load Clarkson Algorithm, every node was successful in sampling a random subset and no v_i satisfies $f(R_i) = f(H)$, then $\mathbb{E}[|\{h \in H(U) \mid h \in B\}|] \geq (2/\sqrt{e})^{T/d}$ after these T iterations.*

Proof. Let $B = \{h_1(B), \dots, h_b(B)\}$, $b \leq d$, and let $p_{i,j}$ be the probability that $f(R_i) < f(R_i \cup \{h_j(B)\})$. If node v_i has chosen some R_i with $f(R_i) < f(H)$, then there must exist an $h_j(B)$ with $f(R_i) < f(R_i \cup \{h_j(B)\})$, which implies that under the condition that $f(R_i) < f(H)$, $\sum_j p_{i,j} \geq 1$. The $p_{i,j}$'s are the same for each v_i since each v_i has the same probability of picking some subset R of $H(U)$ of size $6d^2$. Hence, we can simplify $p_{i,j}$ to p_j and state that $\sum_j p_j \geq 1$. Now, let $p_{j,t}$ be the probability that $f(R) < f(R \cup \{h_j(B)\})$ for a randomly chosen subset R in iteration t, and fix any values for the $p_{j,t}$ so that $\sum_j p_{j,t} \geq 1$ for all j and t. Let $\mu_{j,t}$ be the multiplicity of $h_j(B)$ at the end of round t. Then, for all j, $\mu_{j,0} \geq 1$, and

$$\mathbb{E}[\mu_{j,t+1}] \geq \frac{1 + p_{j,t}}{1 + 1/(2d)} \cdot \mu_{j,t}$$

Hence, $\mathbb{E}[\mu_{j,T}] \geq (\prod_{t=1}^{T}(1+p_{j,t}))/(1+1/(2d))^T$. Since $1+x \geq 2^x$ for all $x \in [0,1]$, it follows that $\prod_t(1+p_{j,t}) \geq \prod_t 2^{p_{j,t}} = 2^{\sum_t p_{j,t}}$. Also, since $\sum_{t=1}^{T}\sum_j p_{j,t} \geq T$, there must be a j^* with $\sum_{t=1}^{T} p_{j^*,t} \geq T/d$. Therefore, there must be a j^* with $\mathbb{E}[\mu_{j^*,T}] \geq 2^{T/d}/(1+1/(2d))^T \geq 2^{T/d}/e^{T/(2d)}$, which completes the proof. ◀

Since $|H(U)|$ is bounded by $O(|H_0|)$ w.h.p., the expected number of $h \in H$ with $h \in B$ should be in $O(|H_0|)$ as well. Due to Lemma 8, this cannot be the case if $T = \Omega(d \log |H_0|)$ is sufficiently large. Thus, the algorithm must terminate within $O(d \log |H_0|) = O(d \log n)$ rounds w.h.p. In order to complete the description of our algorithm, we need distributed algorithms satisfying the following claims:

1. The nodes v_i succeed in sampling subsets R_i uniformly at random in a round, w.h.p., with maximum work $O(d^2 + \log n)$.
2. Once a node v_i has chosen an R_i with $f(R_i) = f(H)$, all nodes are aware of that within $O(\log n)$ communication rounds, w.h.p., so that the Low-Load Clarkson Algorithm can terminate. The maximum work for the termination detection is $O(\log n)$ per round.

The next two subsections are dedicated to these algorithms.

2.1 Sampling Random Subsets

For simplicity, we assume here that every node knows the exact value of $\log n$, but it is easy to see that the sampling algorithm also works if the nodes just know a constant factor estimate of $\log n$, if the constant c used below is sufficiently large.

Each node v_i samples a subset R_i in a way that is as simple as it can possibly get: v_i asks (via pull requests) $s = c(6d^2 + \log n)$ many nodes v_j, selected uniformly and independently at random, to send it a random element in $H(v_j)$, where c is a sufficiently large constant. If v_i doesn't receive at least $6d^2$ distinct elements (where two elements are distinct if they do not represent the *same* copy of some $h \in H$), the sampling fails. Otherwise, v_i selects the first $6d^2$ distinct elements (based on an arbitrary order of the pull requests) for its subset R_i. Certainly, the work for each node is just $O(d^2 + \log n)$.

▶ **Lemma 10.** *For any i, node v_i succeeds in sampling a subset R_i uniformly at random, w.h.p.*

Proof. Suppose that v_i succeeds in receiving k distinct elements in the sampling procedure above. Since the elements in $H(U)$ are distributed uniformly and independently at random among the nodes, every subset R of size k in $H(U)$ has the same probability of representing these k elements. Hence, it remains to show that v_i succeeds in receiving at least $6d^2$ distinct elements w.h.p.

Consider any numbering of the pull requests from 1 to s. For the jth pull request of v_i, two bad events can occur. First of all, the pull request might be sent to a node that does not have any elements. Since $|H(U)| \geq n$, the probability for that is at most $(1-1/n)^n \leq 1/e$. Second, the pull request might return an element that was already returned by one of the prior $j-1$ pull requests. Since this is definitely avoided if the jth pull request selects a node that is different from the nodes selected by the prior $j-1$ pull requests, the probability for that is at most $(j-1)/n$. So altogether, the probability that a pull request fails is at most $1/e + s/n \leq 1/2$.

Now, let the binary random variable X_j be 1 if and only if the jth pull request fails. Since the upper bound of $1/2$ for the failure holds independently of the other pull requests, it holds for any subset $S \subseteq \{1,\ldots,s\}$ that $\mathbb{E}[\prod_{j\in S} X_j] \leq (1/2)^{|S|}$. Hence, Theorem 7 implies that $\sum_j X_j \leq 3s/4$ w.h.p. If c is sufficiently large, then $s/4 \geq 6d^2$, which completes the proof. ◀

Note that our sampling strategy does not reveal any information about which elements are stored in $H(v_i)$, so each element still has a probability of $1/n$ to be stored in $H(v_i)$, which implies that Lemma 5 still holds.

2.2 Termination

For efficiency reasons, the termination check is done *concurrently* with the iterations of the repeat-loop, i.e., instead of waiting for the termination check to complete, the nodes already start a new iteration. We use the following strategy for each node v_i:

Suppose that in iteration t of the repeat loop, $|V_i| = 0$, i.e., $f(R_i) = f(R_i \cup H(v_i))$. Then v_i determines an optimal basis B of R_i, stores the entry $(t, B, 1)$ in a dedicated set S_i, and performs a push operation on $(t, B, 1)$. At the beginning of iteration t_i of the repeat loop, v_i works as described in Algorithm 3. In the comparison between $f(B')$ and $f(B)$ we assume w.l.o.g. that $f(B') = f(B)$ if and only if $B' = B$ (otherwise, we use a lexicographic ordering of the elements as a tie breaker). The parameter c in the algorithm is assumed to be a sufficiently large constant known to all nodes.

Algorithm 3 One iteration of the Termination Algorithm for v_i.

```
 1: for all (t, B, x) received by v_i do                ▷ update best seen base w.r.t. t
 2:     if there is some (t, B', x') in S_i then
 3:         if f(B') > f(B) then discard (t, B, x)
 4:         if f(B') < f(B) then replace (t, B', x') by (t, B, x)
 5:         if f(B') = f(B) then
 6:             replace (t, B', x') by (t, B, min{x, x'})
 7:     else
 8:         add (t, B, x) to S_i
 9: for all (t, B, x) in S_i do                          ▷ check optimality and maturity
10:     if ∃h ∈ H(v_i) : f(B) < f(B ∪ {h}) then x := 0                  ▷ not optimal
11:     if t < t_i − c log n then                 ▷ B is mature (t_i: current iteration)
12:         remove (t, B, x) from S_i
13:         if x = 1 then output f(B), stop
14:     else
15:         push(t, B, x)
```

▶ **Lemma 11.** *If the constant c in the termination algorithm is large enough, it holds w.h.p.: Once a node v_i satisfies $f(R_i) = f(H)$, then all nodes v_j output a value $f(B)$ with $f(B) = f(H)$ after $c \log n$ iterations, and if a node v_i outputs a value $f(B)$, then $f(B) = f(H)$.*

Proof. Using standard arguments, it can be shown that if the constant c is large enough, then for every iteration t, it takes at most $(c/2)\log n$ further iterations, w.h.p., until the basis B with maximum $f(B)$ injected into some S_i at iteration t (which we assume to be unique by using some tie breaking mechanism) is contained in all S_i's. At this point, we have two cases. If $f(B) = f(H)$, then for all v_i, there is no $h \in H(v_i)$ with $f(B) < f(B \cup \{h\})$ at any point from iteration t to $t + (c/2)\log n$, and otherwise, there must be at least one v_i at iteration $t + (c/2)\log n$ with $f(B) < f(B \cup \{h\})$ for some $h \in H(v_i)$. In the first case, no v_i will ever set x in the entry (t, B, x) to 0, so after an additional $(c/2)\log n$ iterations, every v_i

still stores $(t, B, 1)$ and therefore outputs B. In the second case, there is at least one entry of the form $(t, B, 0)$ at iteration $s + (c/2)\log n$. For this entry, it takes at most $(c/2)\log n$ further iterations, w.h.p., to spread to all nodes so that at the end, no node outputs B. ◀

Since the age of an entry is at most $c \log n$ and for each age a node performs at most one push per iteration, every node executes just $O(\log n)$ pushes per iteration.

2.3 Extension to Any $|H| \geq 1$

If $|H| < n$, the probability that our sampling strategy might fail will get too large. Hence, we need to extend the Low-Load Clarkson algorithm so that we quickly reach a point where $|H(U)| \geq n$ at any time afterwards. We do this by integrating a so-called *pull phase* into the algorithm.

Initially, a node v_i sets its Boolean variable *pull* to *true* if and only if $H_0(v) = \emptyset$ (which would happen if none of the elements in H has been assigned to it). Afterwards, it executes the algorithm shown in Algorithm 4. As long as $pull = true$ (i.e., v_i is still in its pull phase), v_i keeps executing a pull operation in each iteration of the algorithm, which asks the contacted node v_j to send it a copy of a random element in $H_0(v_j)$, until it successfully receives an element h that way. Once this is the case, v_i pushes the successfully pulled element to a random node v_j (so that all elements are distributed uniformly and independently at random among the nodes), which will store it in $H_0(v_j)$, and starts executing the Low-Load Clarkson algorithm from above.

Algorithm 4 Extended Low-Load Clarkson Algorithm for v_i.

```
1:  repeat
2:      if pull = true then                                   ▷ v_i is still in its pull phase
3:          pull(h)                                           ▷ v_i expects some h ∈ H_0
4:          if h ≠ NULL then
5:              push(h, 0)                                    ▷ 0: flag that h belongs to H_0
6:              pull := false                                 ▷ pull phase is over
7:      else
8:          choose a random subset R_i of size 6d^2 from H(U)
9:          if the sampling of R_i succeeds then
10:             V_i := {h ∈ H(v_i) | f(R_i) < f(R_i ∪ {h})}
11:             for all h ∈ V_i do push(h)
12:     for all (h, 0) received by v_i do add h to H_0(v_i)
13:     for all h received by v_i do add h to H(v_i)
14:     for all h ∈ H(v_i) − H_0(v_i) do
15:         keep h with probability 1/(1 + 1/(2d))
16: until v_i outputs a solution
```

▶ **Lemma 12.** *After $O(\log n)$ rounds, all nodes have completed their pull phase, w.h.p.*

Proof of Lemma 11. Note that no node will ever delete an element in H_0, and pull requests only generate elements for H_0, so the filtering approach of the Low-Load Clarkson algorithm cannot interfere with the pull phase. Thus, it follows from a slight adaptation of proofs in previous work on gossip algorithms (e.g., [13]) that for any $|H| \geq 1$, all nodes have completed their pull phase after at most $O(\log n)$ rounds, w.h.p. ◀

Certainly, $|H_0| \leq n + |H| = O(n \log n)$ and $H \subseteq H_0$ at any time, and once all nodes have finished their pull phase, $|H_0| \geq n$, so we are back to the situation of the original Low-Load Clarkson Algorithm.

During the time when some nodes are still in their pull phase, some nodes might already be executing Algorithm 2, which may cause the sampling of R_i to fail for some nodes v_i. However, the analyses of Lemma 6 and Lemma 8 already take that into account. Once all nodes have finished their pull phase, Lemma 9 applies, which means that after an additional $O(d \log n)$ rounds at least one node has found the optimal solution, w.h.p. Thus, after an additional $O(\log n)$ rounds, all nodes will know the optimal solution and terminate. Altogether, we therefore still get the same runtime and work bounds as before, completing the proof of Theorem 3.

3 High-Load Clarkson Algorithm

If $|H| = \omega(n \log n)$, then our LP-type algorithm in the previous section will become too expensive since, on expectation, $|V_i|$ would be in the order of $m/(dn)$, which is now $\omega(\log n)$. In this section, we present an alternative distributed LP-type algorithm that just causes $O(d \log n)$ work for any $|H| = poly(n)$, but the internal computations are more expensive than in the algorithm presented in the previous section because now $f(S)$-values for sets of size $\Theta(m/n)$ have to be computed instead of just $O(d^2)$. Again, we assume that initially the elements in H are randomly distributed among the nodes in U. Let the initial $H(v_i)$ be all elements of H assigned that way to v_i. As before, $H(U) = \bigcup_i H(v_i)$.

Algorithm 5 High-Load Clarkson Algorithm.

```
1: repeat
2:     for all nodes v_i in parallel do
3:         compute an optimal basis B_i of H(v_i)
4:         push(B_i)
5:         for all B_j received by v_i do
6:             V_j := {h ∈ H(v_i) | f(B_j) < f(B_j ∪ {h})}
7:             for all h ∈ V_j do push(h)
8:         for all h received by v_i do add h to H(v_i)
9: until at least one v_i satisfies f(H(v_i)) = f(H)
```

Irrespective of which elements get selected for the V_i's in each round, $H(v_i)$ is a random subset of $H(U)$ because the elements in H are assumed to be randomly distributed among the nodes and every element in V_i is sent to a random node in U. Hence, if follows from $|H(U)| = \omega(n \log n)$ and the standard Chernoff bounds that $|H(v_i)|$ is within $(1 \pm \epsilon)|H(U)|/n$, w.h.p., for any constant $\epsilon > 0$. Thus, we are computing bases of random subsets R of size r within $(1 \pm \epsilon)|H(U)|/n$, w.h.p. This, in turn, implies with $\mathbb{E}[|V_i|] \leq d \cdot \frac{m-r}{n(r+1)}$, where $m = |H(U)|$, that $\mathbb{E}[|V_i|] \leq (1 + \epsilon)d$. In the worst case, however, $|V_i|$ could be very large, so just bounding the expectation of $|V_i|$ does not suffice to show that our algorithm has a low work. Therefore, we need a proper extension of Lemma 5 that exploits higher moments. Note that it works for arbitrary LP-type problems, i.e., also problems that are non-regular and/or degenerate.

▶ **Lemma 13.** *Let (H, f) be an LP-type problem of dimension d and let μ be any multiplicity function. For any $k \geq 1$ and any $1 \leq r < m/2 - k$, where $m = |H(\mu)|$, it holds for $V = \{h \in H(\mu) \mid f(R) < f(R \cup \{h\})\}$ for a random subset R of size r from $H(\mu)$ that $\mathbb{E}[|V|^k] \leq 2(k \cdot d \cdot (m-r)/(r+1))^k$.*

Proof of Lemma 12. By definition of the expected value it holds that

$$\mathbb{E}[|V|^k] = \frac{1}{\binom{m}{r}} \sum_{R \in \binom{H(\mu)}{r}} |V_R|^k$$

For $R \in \binom{H(\mu)}{r}$ and $h \in H(\mu)$ let $X(R, h)$ be the indicator variable for the event that $f(R) < f(R \cup \{h\})$. Then we have

$$\begin{aligned}
\binom{m}{r}\mathbb{E}[|V|^k] &= \sum_{R \in \binom{H(\mu)}{r}} |V_R|^k = \sum_{R \in \binom{H(\mu)}{r}} \left(\sum_{h \in H(\mu) - R} X(R,h) \right)^k \\
&\overset{(1)}{\leq} \sum_{R \in \binom{H(\mu)}{r}} \left(\sum_{h \in H(\mu)-R} X(R,h) + 2^k \sum_{\{h_1,h_2\} \subseteq H(\mu)-R} X(R,h_1) \cdot X(R,h_2) + \ldots \right. \\
&\qquad \left. + k^k \sum_{\{h_1,\ldots,h_k\} \subseteq H(\mu)-R} X(R,h_1) \cdot \ldots \cdot X(R,h_k) \right)
\end{aligned}$$

(1) holds because $X(R,h)^i = X(R,h)$ for any $i \geq 1$ and there are at most i^k ways of assigning k $X(R,h)$'s, one from each sum in $(\sum_{h \in H(\mu)-R} X(R,h))^k$, to the i $X(R,h)$'s in some $X(R,h_1) \cdot \ldots \cdot X(R,h_i)$. Moreover, for any $k > 1$,

$$\begin{aligned}
&\sum_{R \in \binom{H(\mu)}{r}} \sum_{\{h_1,\ldots,h_k\} \subseteq H(\mu)-R} X(R,h_1) \cdot \ldots \cdot X(R,h_k) \\
&= \sum_{Q \in \binom{H(\mu)}{r+k}} \sum_{\{h_1,\ldots,h_k\} \subseteq Q} X(Q - \{h_1,\ldots,h_k\}, h_1) \cdot \ldots \cdot X(Q - \{h_1,\ldots,h_k\}, h_k) \\
&= \sum_{Q \in \binom{H(\mu)}{r+k}} \sum_{\{h_1,\ldots,h_{k-1}\} \subseteq Q} \sum_{h_k \in Q - \{h_1,\ldots,h_{k-1}\}} X(Q - \{h_1,\ldots,h_k\}, h_1) \cdot \ldots \\
&\qquad \cdot X(Q - \{h_1,\ldots,h_k\}, h_{k-1}) \cdot X((Q - \{h_1,\ldots,h_{k-1}\}) - h_k, h_k) \\
&\overset{(2)}{\leq} \sum_{Q \in \binom{H(\mu)}{r+k}} \sum_{\{h_1,\ldots,h_{k-1}\} \subseteq Q} \\
&\qquad \sum_{h_k \in B(Q - \{h_1,\ldots,h_{k-1}\})} X((Q - h_k) - \{h_1,\ldots,h_{k-1}\}, h_1) \cdot \ldots \\
&\qquad \qquad \cdot X((Q - h_k) - \{h_1,\ldots,h_{k-1}\}, h_{k-1}) \\
&\leq \sum_{Q \in \binom{H(\mu)}{r+k}} d \cdot \max_{h_k \in Q} \left(\sum_{\{h_1,\ldots,h_{k-1}\} \subseteq Q - h_k} X((Q - h_k) - \{h_1,\ldots,h_{k-1}\}, h_1) \cdot \ldots \right. \\
&\qquad \left. \cdot X((Q - h_k) - \{h_1,\ldots,h_{k-1}\}, h_{k-1}) \right) \\
&\leq \ \ldots \ \leq \sum_{Q \in \binom{H(\mu)}{r+k}} d^k
\end{aligned}$$

where $B(S)$ is an optimal basis of S. (2) holds because $X((Q - \{h_1, \ldots, h_{k-1}\}) - h_k, h_k) = 0$ for every $h_k \notin B(Q - \{h_1, \ldots, h_{k-1}\})$. The skipped calculations apply the same idea for h_k to $h_{k-1}, \ldots, h_2$. Hence, as long as $r + k < |H(\mu)|/2$,

$$\binom{m}{r}\mathbb{E}[|V|^k] = \sum_{R\in\binom{H(\mu)}{r}} |V_R|^k \leq \sum_{Q\in\binom{H(\mu)}{r+1}} d + 2^k \sum_{Q\in\binom{H(\mu)}{r+2}} d^2 + \ldots + k^k \sum_{Q\in\binom{H(\mu)}{r+k}} d^k$$
$$\leq 2k^k \sum_{Q\in\binom{H(\mu)}{r+k}} d^k = 2(dk)^k \binom{m}{r+k}$$

Resolving that to $\mathbb{E}[|V|^k]$ results in the lemma. ◀

Lemma 13 allows us to prove the following probability bound, which is essentially best possible for constant d by a lower bound in [9].

▶ **Lemma 14.** *Let (H, f) be an LP-type problem of dimension d and let μ be any multiplicity function. For any $\gamma \geq 1$ and $1 \leq r < m/2 - \gamma$, where $m = |H(\mu)|$, it holds for $V = \{h \in H(\mu) \mid f(R) < f(R \cup \{h\})\}$ for a random subset R of size r from $H(\mu)$ that*

$$\Pr[|V| \geq 4\gamma \cdot \frac{d \cdot m}{r+1}] \leq 1/2^\gamma$$

Proof. From Lemma 13 and the Markov inequality it follows that, for any $c \geq 1$ and $k \geq 1$, $\Pr[|V|^k \geq c^k \cdot 2(k \cdot d \cdot (m-r)/(r+1))^k] \leq 1/c^k$ and therefore,

$$\Pr[|V| \geq c \cdot (1 + 1/k)(k \cdot d \cdot (m-r)/(r+1))] \leq 1/c^k$$

Setting $c = 2$ and $k = \gamma$ results in the lemma. ◀

Since, for every element $h \in V$, the probability that $h \in H(v_i)$ is equal to $1/n$, it follows that $|V_i| = O(d \log n)$ for every i, w.h.p., so the maximum work needed for pushing some V_i is $O(d \log n)$. Moreover, the size of $H(U)$ after T iterations is at most $|H| + O(Tdn \log n)$, w.h.p. On the other hand, we will show the following variant of Lemma 9.

▶ **Lemma 15.** *Let B be an arbitrary optimal basis of H. As long as no v_i has satisfied $f(H(v_i)) = f(H)$ so far, $\mathbb{E}[|\{h \in H(U) \mid h \in B\}|] \geq 2^{T/d}$ after T iterations of the High-Load Clarkson Algorithm.*

Proof. Let $B = \{h_1(B), \ldots, h_b(B)\}$, $b \leq d$, and let $p_{i,j}$ be the probability that $f(B_i) < f(B_i \cup \{h_j(B)\})$. If $f(B_i) < f(H)$, then there must exist an $h_j(B)$ with $f(B_i) < f(B_i \cup \{h_j(B)\})$, which implies that under the condition that $f(B_i) < f(H)$, $\sum_j p_{i,j} \geq 1$. Let ρ_j be the expected number of duplicates created for some copy of $h_j(B)$. Since the B_i's are sent to nodes chosen uniformly at random, $\rho_j = (1/n)\sum_i p_{i,j}$. Certainly, since $p_{i,j} \in [0, 1]$ for all i, also $\rho_j \in [0, 1]$. Moreover,

$$\sum_j \rho_j = \sum_j (1/n) \sum_i p_{i,j} = (1/n) \sum_i \sum_j p_{i,j} \geq 1$$

Hence, we can use the same arguments as in the proof of Lemma 9, with p_j replaced by ρ_j and without the term $(1 + 1/(2d))$ in the denominator since we do not perform filtering, to complete the proof. ◀

Thus, because $|H(U)| \leq |H| + O(Tdn \log n)$ after T iterations, w.h.p., our algorithm must terminate within $O(d \log |H|) = O(d \log n)$ rounds, w.h.p. For the termination detection, we can again use the algorithm proposed in Section 2.2, which results in an additional work of $O(\log n)$ per round.

3.1 Accelerated High-Load Clarkson Algorithm

If we are willing to spend more work, we can accelerate the High-Load Clarkson Algorithm. Suppose that in Algorithm 5 node v_i does not just push B_i once but C many times. Then the work for that goes up from $O(d)$ to $O(C \cdot d)$, and the maximum work for pushing out the elements of the W_i's is now bounded by $O(C \cdot d \log n)$, w.h.p., which means that after T rounds, $|H(U)|$ is now bounded by $|H| + O(TC \cdot dn \log n)$, w.h.p. Furthermore, we obtain the following result, which replaces Lemma 15.

▶ **Lemma 16.** *Let B be an arbitrary optimal basis of H. As long as no v_i has satisfied $f(H(v_i)) = f(H)$ so far, $\mathbb{E}[|\{h \in H(U) \mid h \in B\}|] \geq (C+1)^{\lfloor T/d \rfloor}$ after T rounds of the High-Load Clarkson Algorithm with parameter C.*

Proof. Recall the definition of ρ_j in the proof of Lemma 15. It now holds that $\rho_j = (C/n)\sum_i p_{i,j}$, which implies that $\rho_j \in [0, C]$ for all j and $\sum_j \rho_j \geq C$. Now, let $\rho_{j,t}$ be the expected number of duplicates created for some copy of $h_j(B)$ in round t, and fix any values of $\rho_{j,t}$ so that $\sum_j \rho_{j,t} \geq C$ and $\rho_j \in [0, C]$ for all j and t. Let $\mu_{j,t}$ be the multiplicity of $h_j(B)$ at the end of round t. Then, for all j, $\mu_{j,0} \geq 1$, and

$$\mathbb{E}[\mu_{j,t+1}] \geq (1 + \rho_{j,t}) \cdot \mu_{j,t}$$

Hence, $\mathbb{E}[\mu_{j,T}] \geq \prod_{t=1}^{T}(1+\rho_{j,t})$. Suppose that $\sum_{t=1}^{T} \rho_{j,t} = M$. Since $\prod_{t=1}^{T}(1+\rho_{j,t})$ is a convex function (i.e., it attains its maximum when $\rho_{j,t} = \rho_{j,t'}$ for all t, t' under the constraint that $\sum_{t=1}^{T} \rho_{j,t}$ is fixed, which can be seen from the fact that $((1+r)+\epsilon)((1+r)-\epsilon) = (1+r)^2 - \epsilon^2$), it gets lowest if as many of the $\rho_{j,t}$'s are as large as possible and the rest is 0. Thus, $\prod_{t=1}^{T}(1+\rho_{j,t}) \geq (C+1)^{\lfloor M/C \rfloor}$.

Since $\sum_{t=1}^{T}\sum_j p_{j,t} \geq C \cdot T$, there must be a j^* with $\sum_{t=1}^{T} p_{j^*,t} \geq C \cdot T/d$. Therefore, there must be a j^* with $\mathbb{E}[\mu_{j^*,T}] \geq (C+1)^{\lfloor T/d \rfloor}$, which completes the proof. ◀

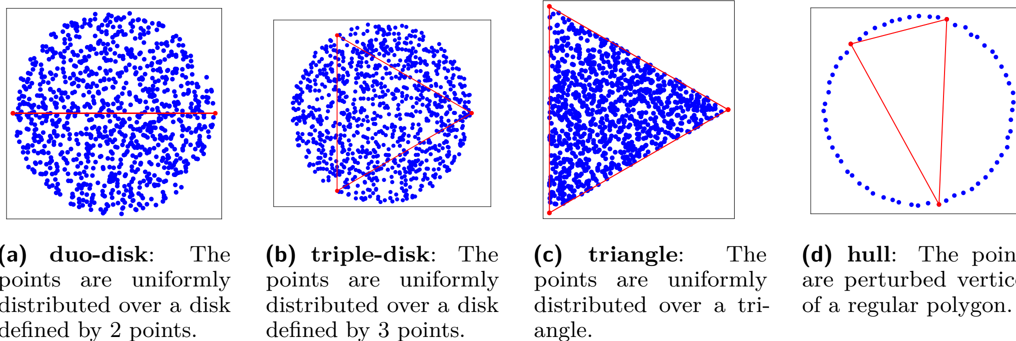
Setting $C = \log^\epsilon n$ for any constant $\epsilon > 0$, it follows that our algorithm must terminate in $O((d/\epsilon)\log(|H|)/\log\log(n)) = O(d\log(n)/\log\log(n))$ rounds, w.h.p. This completes the proof of Theorem 4.

4 Experimental Results

While we have obtained the theoretical bound of $O(d \log n)$ rounds w.h.p. for Algorithms 2 and 5, we are also interested in their practical performance. In particular, we would like to estimate the constant factor hidden in our asymptotic bound. To achieve this, we will look at the specific LP-type problem of finding the minimum enclosing disk, i.e., the two-dimensional version of the minimum enclosing ball problem mentioned in the introduction.

Note that the running time for the termination phase (Algorithm 3) of these algorithms is predictable and independent of the actual input, so we will measure the number of rounds until at least one node found the solution. We consider the four different test cases shown in Figure 1. For each test case, we take the average result of 10 runs of our algorithms with n nodes on n data-points, where $n = 2^i$ ranges over $i = 1, \ldots 14$, (this is extended to 17 for the duo-disk case for the Low-Load Clarkson Algorithm), see Figures 2a and 2b for the results.

For the Low-Load Algorithm, note that the small test cases finish within one round, because there is a high probability that there is a node v_i where $H(v_i)$ contains an optimal basis. For the duo-disk test case the number of rounds is $1.2 \log n$, while it is $1.7 \log n$ for



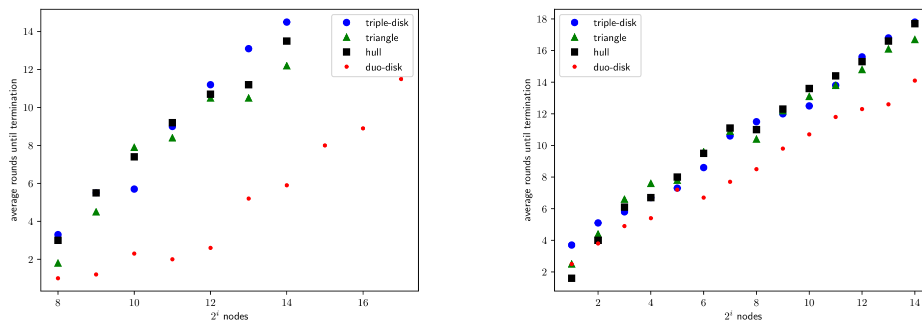
(a) duo-disk: The points are uniformly distributed over a disk defined by 2 points.

(b) triple-disk: The points are uniformly distributed over a disk defined by 3 points.

(c) triangle: The points are uniformly distributed over a triangle.

(d) hull: The points are perturbed vertices of a regular polygon.

Figure 1 The 4 types of data-sets of the minimum enclosing disk problem used in our experimental evaluation: duo-disk, triple-disk, triangle, and hull.



(a) The average number of rounds until a node finds the minimum enclosing disk over 10 runs of the Low-Load Clarkson Algorithm. Test instances of size $< 2^8$ finish in one round.

(b) The average number of rounds until a node finds the minimum enclosing disk over 10 runs of the High-Load Clarkson Algorithm.

Figure 2 The result of the experimental analysis.

the other test cases. For the High-Load Algorithm, the runtime of the duo-disk test cases is around $0.9 \log n$, while it is $1.1 \log n$ for the other test cases. So these experiments show that when applied to the minimum enclosing disk problem, where $d \leq 3$, the constants hidden in our asymptotic bounds are small. Note that the three test cases other than duo-disk behave similarly, while duo-disk runs a bit faster. The difference between the duo-disk case and the other test cases is the size of the optimal basis, which is 2 for duo-disk and 3 for the others. This confirms the runtime bound implied by Lemma 2, since we can define d there as the minimum size of an optimal basis, and suggests that other features of the problem do not influence the number of rounds much.

5 Conclusion

In this paper we presented various efficient distributed algorithms for LP-type problems in the gossip model. Of course, it would be interesting to find out which other problems can be efficiently solved within Clarkson's framework, and whether some of our bounds can be improved.

References

1. Noga Alon and Nimrod Megiddo. Parallel linear programming in fixed dimension almost surely in constant time. *Journal of the ACM*, 41(2):422–434, 1994.
2. Kenneth L. Clarkson. Las Vegas algorithms for linear and integer programming when the dimension is small. *Journal of the ACM*, 42(2):488–499, 1995.
3. Benjamin Doerr, Leslie Ann Goldberg, Lorenz Minder, Thomas Sauerwald, and Christian Scheideler. Stabilizing Consensus with the Power of Two Choices. In *Proc. 23rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 149–158, 2011.
4. Martin Dyer. A parallel algorithm for linear programming in fixed dimension. In *Proc. 11th Symposium on Computational Geometry (SoCG)*, pages 345–349, 1995.
5. Martin Dyer, Bernd Gärtner, Nimrod Megiddo, and Emo Welzl. Linear Programming. In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry, 3rd edition*, chapter 49, pages 49:1–49:19. Chapman and Hall/CRC, 2017.
6. Martin E. Dyer and Alan M. Frieze. A randomized algorithm for fixed-dimensional linear programming. *Mathematical Programming*, 44(1–3):203–212, 1989.
7. Bernd Gärtner. A subexponential algorithm for abstract optimization problems. *SIAM Journal on Computing*, 24(5):1018–1035, 1995.
8. Bernd Gärtner and Emo Welzl. Linear programming – randomization and abstract frameworks. In *Proc. 13th Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 669–687, 1996.
9. Bernd Gärtner and Emo Welzl. A simple sampling lemma: Analysis and applications in geometric optimization. *Discrete Computational Geometry*, 25:569–590, 2001.
10. Michael T. Goodrich. Geometric partitioning made easier, even in parallel. In *Proc. 9th ACM Symposium on Computational Geometry (SoCG)*, pages 73–82, 1993.
11. Bernhard Häupler. Analyzing network coding (gossip) made easy. *Journal of the ACM*, 63(3):26:1–26:22, 2016.
12. Bernhard Häupler, Jeet Mohapatra, and Hsin-Hao Su. Optimal gossip algorithms for exact and approximate quantile computations. In *Proc. ACM Symposium on Principles of Distributed Computing (PODC)*, pages 179–188, 2018.
13. Richard M. Karp, Christian Schindelhauer, Scott Shenker, and Berthold Vöcking. Randomized rumor spreading. In *Proc. 41st IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 565–574, 2000.
14. David Kempe, Alin Dobra, and Johannes Gehrke. Gossip-Based Computation of Aggregate Information. In *Proc. 44th IEEE Symposium on Foundations of Computer Science (FOCS 2003)*, pages 482–491, 2011.
15. Nick Littlestone. Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. In *Proc. 28th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 68–77, 1987.
16. Abhiram G. Ranade. How to emulate shared memory. *Journal of Computer and System Sciences*, 42(3):307–326, 1991.
17. Jeanette Schmidt, Alan Siegel, and Aravind Srinivasan. Chernoff-Hoeffding bounds for applications with limited independence. *SIAM Journal on Discrete Mathematics*, 8(2):223–250, 1995.
18. Micha Sharir and Emo Welzl. A combinatorial bound for linear programming and related problems. In *Proc. 9th Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 569–579, 1992.
19. Petr Škovroň. *Abstract Models of Optimization Problems.* PhD thesis, Charles University, Prague, 2007.
20. Emo Welzl. Partition trees for triangle counting and other range searching problems. In *Proc. 4th ACM Symposium on Computational Geometry (SoCG)*, pages 23–33, 1988.

Privatization-Safe Transactional Memories

Artem Khyzha
Tel Aviv University, Tel Aviv, Israel

Hagit Attiya
Technion – Israel Institute of Technology, Haifa, Israel

Alexey Gotsman
IMDEA Software Institute, Madrid, Spain

Abstract

Transactional memory (TM) facilitates the development of concurrent applications by letting the programmer designate certain code blocks as atomic. Programmers using a TM often would like to access the same data both inside and outside transactions, and would prefer their programs to have a *strongly atomic* semantics, which allows transactions to be viewed as executing atomically with respect to non-transactional accesses. Since guaranteeing such semantics for arbitrary programs is prohibitively expensive, researchers have suggested guaranteeing it only for certain *data-race free (DRF)* programs, particularly those that follow the *privatization* idiom: from some point on, threads agree that a given object can be accessed non-transactionally.

In this paper we show that a variant of *Transactional DRF (TDRF)* by Dalessandro et al. is appropriate for a class of *privatization-safe* TMs, which allow using privatization idioms. We prove that, if such a TM satisfies a condition we call *privatization-safe opacity* and a program using the TM is TDRF under strongly atomic semantics, then the program indeed has such semantics. We also present a method for proving privatization-safe opacity that reduces proving this generalization to proving the usual opacity, and apply the method to a TM based on two-phase locking and a privatization-safe version of TL2. Finally, we establish the inherent cost of privatization-safety: we prove that a TM cannot be progressive and have invisible reads if it guarantees strongly atomic semantics for TDRF programs.

2012 ACM Subject Classification Theory of computation → Concurrency; Theory of computation → Program semantics; Software and its engineering → Software verification

Keywords and phrases Transactional memory, privatization, observational refinement

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.24

Related Version An extended version is available at `https://arxiv.org/abs/1908.03179`.

Funding This research was funded in part by the Israel Science Foundation (grants 2005/17, 1749/14 and 380/18) and the European Research Council (Starting Grant RACCOON).

1 Introduction

Transactional memory (TM) facilitates the development of concurrent applications by letting the programmer designate certain code blocks as *atomic* [23]. TM allows developing a program and reasoning about its correctness as if each atomic block executes as a *transaction* – atomically and without interleaving with other blocks – even though in reality the blocks can be executed concurrently. A TM can be implemented in hardware [24, 29], software [34] or a combination of both [13, 28].

Often programmers using a TM would like to access the same data both inside and outside transactions. This may be desirable to avoid performance overheads of transactional accesses, to support legacy code, or for explicit memory deallocation. One typical pattern is *privatization* [31, 35], illustrated in Figure 1. There the `atomic` blocks return a value signifying whether the transaction committed or aborted. In the program, an object `x` is

© Artem Khyzha, Hagit Attiya, and Alexey Gotsman;
licensed under Creative Commons License CC-BY

```
{ priv = false ∧ x = 0 }
l1 = atomic {                 ||  atomic {
  priv = true; } // T1        ||    if (!priv)
if (l1 == committed)          ||      x = 42;
  x = 1; // n                 ||  } // T2
{ l1 = committed ⟹ x = 1 }
```

Figure 1 Privatization.

```
{ priv = true ∧ x = l = 0 }
x = 42; // n                  ||  l2 = atomic {
l1 = atomic {                 ||    if (!priv)
  priv = false;               ||      l = x;
} // T1                       ||  } // T2
{ l2 = committed ∧ l ≠ 0 ⟹ l = 42 }
```

Figure 2 Publication.

```
{ x = y = 0 }
atomic {       ||
  x = 1;       ||  l1 = x; // n1
  y = 2;       ||  l2 = y; // n2
} // T         ||
{ l1 = 1 ⟹ l2 = 2 }
```

Figure 3 Data race.

guarded by a flag `priv`, showing whether the object should be accessed transactionally (`false`) or non-transactionally (`true`). The left-hand-side thread first tries to set the flag inside transaction T_1, thereby *privatizing* `x`. If successful, it then accesses `x` non-transactionally. A concurrent transaction T_2 in the right-hand-side thread checks the flag `priv` prior to accessing `x`, to avoid simultaneous transactional and non-transactional access to the object. We expect the postcondition shown to hold: if privatization is successful, at the end of the program `x` should store 1, not 42. The opposite idiom is *publication*, illustrated in Figure 2. The left-hand-side thread writes to `x` non-transactionally and then clears the flag `priv` inside transaction T_1, thereby *publishing* `x`. The right-hand-side thread tests the flag inside transaction T_2, and if it is cleared, reads `x`. Again, we expect the postcondition to hold: if the right-hand-side thread sees the write to the flag, it should also see the write to `x`. The two idioms can be combined: the programmer may privatize an object, then access it non-transactionally, and finally publish it back for transactional access.

Ideally, programmers mixing transactional and non-transactional accesses to objects would like their programs to have *strongly atomic semantics* [8], where transactions can be viewed as executing atomically also with respect to non-transactional accesses, i.e., without interleaving with them. This is equivalent to considering every non-transactional access as a single-instruction transaction. For example, the program in Figure 3 under strongly atomic semantics can only produce executions where each of the non-transactional accesses n_1 and n_2 executes either before or after the transaction T, so that the postcondition in Figure 3 always holds. Unfortunately, providing such semantics in software requires instrumenting non-transactional accesses with additional instructions for maintaining TM metadata [19]. This undermines scalability and makes it difficult to reuse legacy code. Since most existing TMs are either software-based or rely on a software fall-back, they do not perform such instrumentation and, hence, provide weaker atomicity guarantees. For example, they may allow the program in Figure 3 to execute non-transactional accesses n_1 and n_2 between transactional writes to `x` and `y` and, thus, observe an intermediate state of the transaction, e.g., $\texttt{x} = 1$ and $\texttt{y} = 0$, violating the postcondition in Figure 3.

Researchers have suggested resolving the tension between strong TM semantics and performance by guaranteeing strongly atomic semantics only to *data-race free (DRF)* programs – informally, programs without concurrent transactional and non-transactional accesses to the same data [4, 5, 10, 11, 31, 33, 35]. For example, we do not have to guarantee strongly atomic semantics for the program in Figure 3, which has such concurrent accesses to `x` and `y`. On the other hand, the programs in Figure 1 and Figure 2 should be guaranteed strongly atomic semantics, since at any point of time, an object is accessed either only transactionally or only non-transactionally. Despite the intuitive simplicity of this idea, coming up with a precise DRF definition is nontrivial: early on there were multiple competing proposals for the notion of DRF, and it was unclear how to select among them [4, 10, 11, 25, 30]. To address this, we have recently formalized the requirements on an appropriate notion of DRF using observational refinement [27]: a TM needs to guarantee that, if a program is DRF under

the strongly atomic semantics (formalized as *transactional sequential consistency* [11]), then all its executions are observationally equivalent to strongly atomic ones. This *Fundamental Property* allows the programmer to never reason about weakly atomic semantics at all, even when checking DRF.

Different TMs have different requirements on mixing transactional and non-transactional accesses needed to validate the Fundamental Property. *Privatization-safe* TMs, such as lock-based TMs [15, 21] and NOrec [12], allow the programmer to ensure the absence of concurrent transactional and non-transactional accesses by synchronizing them using transactional operations. Then the program in Figure 1, which synchronizes accesses to `x` using `priv`, is guaranteed strongly atomic semantics as is. *Privatization-unsafe* TMs, such as TL2 [14] and TinySTM [16], require the programmer to insert additional synchronization, e.g., via *transactional fences* [31, 35], which block until all the transactions that were active when the fence was invoked complete. For example, such TMs do not guarantee strongly atomic semantics to the program in Figure 1 unless the transaction T_1 is immediately followed by a transactional fence. This is because TMs such as TL2 execute transactions optimistically, flushing their writes to memory only on commit. Then, in the absence of a fence, the transaction T_1 can privatize `x` and n can modify it after T_2 started committing, but before its write to `x` reached the memory, so that T_2's write subsequently overwrites n's write and violates the postcondition. TMs that make transactional updates in-place and undo them on abort are subject to a similar problem.

Privatization-safe TMs provide a simpler programming model, since they do not require the programmer to select where to place fences. However, the programmer still needs to avoid programs of the kind shown in Figure 3, which would lead the TM to violate strong atomicity. In this paper we show that a variant of *transactional DRF (TDRF)* previously proposed by Dalessandro et al. [11] is appropriate to formalize the programmer's obligations. To this end, we show that this variant of TDRF validates the Fundamental Property, provided the TM satisfies a generalization of *opacity* [20, 21], which we call *privatization-safe opacity*. To formulate this kind of opacity, we generalize TDRF to arbitrary TM histories, not just strongly atomic ones. These results complement our previous proposal of DRF for privatization-unsafe TMs, which considers a more low-level programming model requiring fence placements [27].

We furthermore present a method for proving privatization-safe opacity and apply it to a TM based on two-phase locking [21] and a privatization-safe version of TL2 [14] that executes a fence at the end of each transaction. A key feature of our method is that it reduces proving privatization-safe opacity to proving the ordinary opacity of the TM assuming no mixed transactional/non-transactional accesses. This allows us to reuse the previous proofs of opacity of the two-phase locking TM [21] and TL2 [27].

Finally, our framework allows proving an interesting result about the inherent cost of privatization-safety. We prove that a TM that provides strongly atomic semantics to TDRF programs cannot be *progressive* and have *invisible reads*: it cannot ensure that transactions always complete when running solo and also that transactions reading objects do not prevent transactions writing to them from committing. This result significantly simplifies and strengthens a lower bound by Attiya and Hillel [7], which did not use a formal DRF notion.

2 Programming Language and Strongly Atomic Semantics

Language syntax. We formalize our results for a simple programming language with mixed transactional and non-transactional accesses. A *program* $P = C_1 \parallel \ldots \parallel C_N$ in our language is a parallel composition of *commands* C_t executed by different *threads* $t \in \mathsf{ThreadID} =$

$\{1, \ldots, N\}$. Every thread $t \in \mathsf{ThreadID}$ has a set of *local variables* $l \in \mathsf{LVar}_t$, which only it can access; for simplicity, we assume that these are integer-valued. Threads have access to a *transactional memory* (*TM*), which manages a fixed collection of *shared register objects* $x \in \mathsf{Reg}$. The syntax of commands $C \in \mathsf{Com}$ is as follows:

$$C ::= c \mid C\,;C \mid \ \texttt{if}\,(b)\ \texttt{then}\ C\ \texttt{else}\ C \mid \ \texttt{while}\,(b)\ \texttt{do}\ C$$
$$\mid l = \texttt{atomic}\,\{C\} \mid l = x.\texttt{read}() \mid x.\texttt{write}(e)$$

where b and e denote Boolean, respectively, integer *expressions* over local variables and constants. The language includes *primitive commands* $c \in \mathsf{PCom}$, which operate on local variables, and standard control-flow constructs. An *atomic block* $l = \texttt{atomic}\,\{C\}$ executes C as a *transaction*, which the TM can *commit* or *abort*. The system's decision is returned in the local variable l, which receives a distinguished value committed or aborted. We do not allow programs to abort a transaction explicitly and forbid nested atomic blocks. Threads can invoke two methods on a register x: $x.\texttt{read}()$ returns the current value of x, and $x.\texttt{write}(e)$ sets it to e. These methods may be invoked both *inside* and *outside* atomic blocks.

Model of computations. The semantics of our programming language is defined in terms of *traces* – certain finite sequences of *actions*, each describing a single computation step (in this paper we consider only finite computations). Let ActionId be a set of *action identifiers*. Actions are of two kinds. A *primitive action* denotes the execution of a primitive command and is of the form (a, t, c), where $a \in \mathsf{ActionId}$, $t \in \mathsf{ThreadID}$ and $c \in \mathsf{PCom}$. An *interface action* has one of the following forms (where $x \in \mathsf{Reg}$ and $v \in \mathbb{Z}$):

Request actions	Matching response actions
$(a, t, \mathsf{begintx})$	$(a, t, \mathsf{ok}) \mid (a, t, \mathsf{aborted})$
$(a, t, \mathsf{trycommit})$	$(a, t, \mathsf{committed}) \mid (a, t, \mathsf{aborted})$
$(a, t, \mathsf{write}(x, v))$	$(a, t, \mathsf{ret}(\bot)) \mid (a, t, \mathsf{aborted})$
$(a, t, \mathsf{read}(x))$	$(a, t, \mathsf{ret}(v)) \mid (a, t, \mathsf{aborted})$

Interface actions usually denote the control flow of a thread t crossing the boundary between the program and the TM: *request* actions correspond to the control being transferred from the former to the latter, and *response* actions, the other way around. A begintx action is generated upon entering an `atomic` block, and a trycommit action when a transaction tries to commit upon exiting an `atomic` block. The request actions $\mathsf{write}(x, v)$ and $\mathsf{read}(x)$ denote invocations of the `write`, respectively, `read` methods of register x; a `write` action is annotated with the value v written. The response actions $\mathsf{ret}(\bot)$ and $\mathsf{ret}(v)$ denote the return from invocations of `write`, respectively, `read` methods of a register; the latter is annotated with the value v read. The TM may abort a transaction at any point when it is in control; this is recorded by an aborted response action. To simplify notation, we reuse the interface actions for reads and writes to denote accesses outside transactions.

A *trace* τ is a finite sequence of actions satisfying the expected well-formedness conditions, e.g., that request and response actions are properly matched, and so are actions denoting the beginning and the end of transactions (we defer the formal definition to [26, §A]). A *transaction* T is a nonempty trace such that it contains actions by the same thread, begins with a begintx action and only its last action can be a committed or an aborted action. A transaction T is: *committed* if it ends with a committed action, *aborted* if it ends with aborted, *commit-pending* if it ends with trycommit, and *live*, in all other cases. A transaction T is in a trace τ if T is a subsequence of τ and no longer transaction is. We refer to interface actions in a trace outside of a transaction as *non-transactional actions*. We call a matching request/response pair of a read or a write a *non-transactional access* (ranged over by n).

A *history* is a trace containing only interface actions (thus, omitting all accesses to local variables); we use H, S to range over histories, and $H(i)$ to refer to the i-th action in H. We also use $\mathsf{history}(\tau)$ to denote a projection of a trace to interface actions. Since histories fully capture the possible interactions between a TM and a client program, we often conflate the notion of a TM and the set of histories it produces. Hence, a *transactional memory* $\mathcal{H}$ is a prefix-closed set of histories. We assume that a TM always allows a client program to execute a request and, hence, require $\mathcal{H}$ to be closed under appending any request action to its histories, provided that the latter remain well-formed. Note that histories include actions corresponding to non-transactional accesses, even though these may not be directly managed by the TM implementation. This is needed to account for changes to registers performed by such actions when defining the TM semantics: e.g., in the case when a register is privatized, modified non-transactionally and then published back for transactional access. Of course, a well-formed TM semantics should not impose restrictions on the placement of non-transactional actions, since these are under the control of the program.

Strongly atomic semantics. The *semantics* of a program P is given by the set $[\![P]\!](\mathcal{H})$ of traces it produces when executed with a TM $\mathcal{H}$. Its formal definition follows the intuitive meaning of commands, and we defer it to [26, §A]. Our semantics assumes that the underlying memory is sequentially consistent, which allows us to focus on the key issues specific to TM (we leave handling weak memory for future work, discussed in §9). We use the semantics instantiated with one particular TM to define the *strongly atomic* semantics of programs [8], which is equivalent to transactional sequential consistency [11]. Following [6], we use an *atomic* TM $\mathcal{H}_{\mathsf{atomic}}$ for this purpose: the strongly atomic semantics of a program P is given by the set of traces $[\![P]\!](\mathcal{H}_{\mathsf{atomic}})$. The TM $\mathcal{H}_{\mathsf{atomic}}$ contains only histories that are *non-interleaved*, i.e., where actions by one transaction do not overlap with actions of another transaction or of non-transactional accesses. Out of such histories, $\mathcal{H}_{\mathsf{atomic}}$ contains only histories following the intuitive atomic semantics of transactions: every response action of a $\mathsf{read}(x)$ returns the value v in the last preceding $\mathsf{write}(x, v)$ action that is not located in an aborted or live transaction different from the one of the read; if there is no such write, the read returns the initial value v_{init}. We defer a formal definition of $\mathcal{H}_{\mathsf{atomic}}$ to [26, §A].

3 Transactional Data-Race Freedom

We now formalize in our framework a variant of *transactional data-race freedom (TDRF)* of Dalessandro et al. [11]. According to this notion, a data race happens between a pair of *conflicting* actions, as defined below.

▶ **Definition 1.** *A non-transactional request action α and a transactional request action α'* conflict *if α and α' are executed by different threads, they are read or write actions on the same register, and at least one of them is a write.*

As is standard, we formalize when conflicting actions form a data race using a *happens-before* relation $\mathsf{hb}(H)$ on actions in a history H. We first define the *execution order* of H as follows: $\alpha <_H \alpha'$ iff for some i and j, $\alpha = H(i)$, $\alpha' = H(j)$ and $i < j$.

▶ **Definition 2.** *The* happens-before *relation of a history $H \in \mathcal{H}_{\mathsf{atomic}}$ is* $\mathsf{hb}(H) \triangleq (\mathsf{po}(H) \cup \mathsf{ef}(H) \cup \mathsf{cl}(H))^+$, *where*

- per-thread order $\mathsf{po}(H)$*: $\alpha <_{\mathsf{po}(H)} \alpha'$ iff $\alpha <_H \alpha'$ and α, α' are by the same thread;*
- effect order $\mathsf{ef}(H)$*: $\alpha <_{\mathsf{ef}(H)} \alpha'$ iff $\alpha <_H \alpha'$ and α, α' are by different transactions;*
- client order $\mathsf{cl}(H)$*: $\alpha <_{\mathsf{cl}(H)} \alpha'$ iff $\alpha <_H \alpha'$ and α, α' are non-transactional in H.*

▶ **Definition 3.** *A history $H \in \mathcal{H}_{\mathsf{atomic}}$ is* transactional data-race free*, written* $\mathsf{TDRF}(H)$*, if every pair of conflicting actions in it is ordered by* $\mathsf{hb}(H)$ *one way or another. A program P is* transactional data-race free*, written* $\mathsf{TDRF}(P)$*, if* $\forall \tau \in [\![P]\!](\mathcal{H}_{\mathsf{atomic}}).\, \mathsf{TDRF}(\mathsf{history}(\tau))$.

Components of happens-before used to define TDRF describe various forms of synchronization available in our programming language. First, actions by the same thread cannot be concurrent and thus we let $\mathsf{po}(H) \subseteq \mathsf{hb}(H)$. Second, privatization-safe TMs provide synchronization between transactions, which follows their order in non-interleaved histories of an atomic TM considered in the definition of TDRF on programs. Thus, we let $\mathsf{ef}(H) \subseteq \mathsf{hb}(H)$. Finally, we let $\mathsf{cl}(H) \subseteq \mathsf{hb}(H)$, because in this paper we assume a sequentially consistent memory model and, hence, do not consider pairs of conflicting non-transactional accesses as races. This is the key difference between our variant of TDRF and the original definition by Dalessandro et al. [11], which does not include the client order into happens-before. Our variant of TDRF imposes fewer obligations on the programmer: as we show by establishing the Fundamental Property for our variant of TDRF (§5), under sequentially consistent memory races on non-transactional accesses are harmless for privatization-safety.

To illustrate the TDRF definition, we show that the program in Figure 1 is TDRF by considering the histories it produces with the atomic TM (the program in Figure 2 can be shown TDRF analogously). The possible conflicts are between the accesses to `x` in n and T_2. For a conflict to occur, T_2 has to read `false` from `priv`; then T_2 has to execute before T_1, yielding a history of the form $T_2\, T_1\, n$. In this history T_2 precedes T_1 in the effect order and T_1 precedes n in the per-thread order, meaning that $\mathsf{hb}(H)$ orders the conflict between T_2 and n. Similarly, in [26, §B] we show that programs following a *proxy privatization* pattern [37], where one thread privatizes an object for another thread, are also TDRF. On the other hand, the program in Figure 3 is not TDRF, since in histories it produces with the atomic TM, the happens-before never relates T with n_1 and n_2. Finally, the inclusion of $\mathsf{cl}(H) \subseteq \mathsf{hb}(H)$ allows us to consider DRF those programs that privatize an object by agreeing on its status outside transactions ("partitioning by consensus" in [35]); we provide an example in [26, §B].

4 Privatization-Safe Opacity

We now present our first contribution – a generalization of *opacity* of a TM $\mathcal{H}$ [20, 21] that guarantees that the TM provides strongly atomic semantics to TDRF programs. We call this generalization *privatization-safe opacity*. Its definition requires that a history H of a TM $\mathcal{H}$ can be matched by a history S of the atomic TM $\mathcal{H}_{\mathsf{atomic}}$ that "looks similar" to H from the perspective of the program. The similarity is formalized by a relation $H \sqsubseteq S$, which requires S to be a permutation of H preserving its per-thread and client orders.

▶ **Definition 4.** *A history H_1 corresponds to a history H_2, written $H_1 \sqsubseteq H_2$, if there is a bijection $\theta : \{1, \ldots, |H_1|\} \to \{1, \ldots, |H_2|\}$ such that $\forall i.\, H_1(i) = H_2(\theta(i))$ and*

$$\forall i, j.\, i < j \wedge H_1(i) <_{\mathsf{po}(H_1) \cup \mathsf{cl}(H_1)} H_2(j) \implies \theta(i) < \theta(j).$$

The above relation differs in several ways from the one used to define the ordinary opacity. First, unlike in the ordinary opacity, our histories include non-transactional actions, because these can affect the behavior of the TM. Second, instead of preserving $\mathsf{cl}(H_1)$ in Definition 4, the ordinary opacity requires preserving the following *real-time order* $\mathsf{rt}(H_1)$ on actions: $\alpha <_{\mathsf{rt}(H)} \alpha'$ iff $\alpha \in \{(_, _, \mathsf{committed}), (_, _, \mathsf{aborted})\}$, $\alpha' = (_, _, \mathsf{begintx})$ and $\alpha <_H \alpha'$. This orders non-overlapping transactions, with the duration of a transaction determined by the interval from its begintx action to the corresponding committed or aborted action (or to the end of the history if there is none). However, preserving real-time order is unnecessary if all means of communication between program threads are reflected in histories [17].

We next lift privatization-safe opacity to TMs. A straightforward definition, mirroring the ordinary opacity, would require any history of the TM $\mathcal{H}$ to have a matching history of the atomic TM $\mathcal{H}_{\mathsf{atomic}}$. However, such a requirement would be too strong for our setting: since the TM has no control over non-transactional actions of its clients, histories in $\mathcal{H}$ may be produced by racy programs, and we do not want to require the TM to guarantee strong atomicity in such cases. For example, even though a simple TM based on a single global lock is privatization-safe, it has a history produced by the program from Figure 3 that does not have a matching history of $\mathcal{H}_{\mathsf{atomic}}$ (§1). Hence, our definition of privatization-safe opacity requires only histories produced by TDRF programs to have justifications in $\mathcal{H}_{\mathsf{atomic}}$. To express this restriction, we generalize data-race freedom to be defined over an arbitrary concurrent history H, not just one produced by $\mathcal{H}_{\mathsf{atomic}}$. The new DRF requires that every history of the atomic TM matching H according to the opacity relation be TDRF.

▶ **Definition 5.** *A history $H \in \mathcal{H}$ is* concurrent data-race free*, written* $\mathsf{CDRF}(H)$*, if* $\forall S \in \mathcal{H}_{\mathsf{atomic}}.\, H \sqsubseteq S \implies \mathsf{TDRF}(S)$*. Let* $\mathcal{H}|_{\mathsf{CDRF}} = \{H \in \mathcal{H} \mid \mathsf{CDRF}(H)\}$*. A program P is concurrent data-race free with a TM $\mathcal{H}$, written* $\mathsf{CDRF}(P, \mathcal{H})$*, if* $\forall \tau \in [\![P]\!](\mathcal{H}).\, \mathsf{CDRF}(\mathsf{history}(\tau))$*.*

▶ **Definition 6.** *A TM $\mathcal{H}$ is* privatization-safe opaque*, written* $\mathcal{H}|_{\mathsf{CDRF}} \sqsubseteq \mathcal{H}_{\mathsf{atomic}}$*, if for every history $H \in \mathcal{H}|_{\mathsf{CDRF}}$ there exists a history $S \in \mathcal{H}_{\mathsf{atomic}}$ such that $H \sqsubseteq S$ holds.*

The following lemma (proved in [26, §C]) justifies using CDRF as a generalization of TDRF to concurrent histories by establishing that TDRF programs indeed produce CDRF histories.

▶ **Lemma 7.** *For every program P and a TM system $\mathcal{H}$,* $\mathsf{TDRF}(P)$ *implies* $\mathsf{CDRF}(P, \mathcal{H})$*.*

5 The Fundamental Property

We next formalize the Fundamental Property of TDRF using *observational refinement* [6]: if a program is TDRF under the atomic TM $\mathcal{H}_{\mathsf{atomic}}$, then any trace of the program under a privatization-safe opaque TM $\mathcal{H}$ has an *observationally equivalent* trace under $\mathcal{H}_{\mathsf{atomic}}$.

▶ **Definition 8.** *Traces τ and τ' are* observationally equivalent*, denoted by $\tau \sim \tau'$, if $\forall t.\, \tau|_t = \tau'|_t$ and $\tau|_{\mathsf{nontx}} = \tau'|_{\mathsf{nontx}}$, where $\tau|_{\mathsf{nontx}}$ denotes the subsequence of τ containing all actions from non-transactional accesses.*

Equivalent traces are considered indistinguishable to the user. In particular, the sequences of non-transactional accesses in equivalent traces (which usually include all I/O) satisfy the same linear-time temporal properties. We lift the equivalence to sets of traces as follows.

▶ **Definition 9.** *A set of traces $\mathcal{T}$* observationally refines *a set of traces $\mathcal{T}'$, written $\mathcal{T} \preceq \mathcal{T}'$, if $\forall \tau \in \mathcal{T}.\, \exists \tau' \in \mathcal{T}'.\, \tau \sim \tau'$.*

▶ **Theorem 10** (Fundamental Property)**.** *If $\mathcal{H}$ is a TM such that $\mathcal{H}|_{\mathsf{CDRF}} \sqsubseteq \mathcal{H}_{\mathsf{atomic}}$ and P is a program such that* $\mathsf{TDRF}(P)$*, then* $[\![P]\!](\mathcal{H}) \preceq [\![P]\!](\mathcal{H}_{\mathsf{atomic}})$*.*

Theorem 10 establishes a contract between the programmer and the TM implementors. The TM implementor has to ensure privatization-safe opacity of the TM assuming the program is DRF: $\mathcal{H}|_{\mathsf{CDRF}} \sqsubseteq \mathcal{H}_{\mathsf{atomic}}$. The programmer has to ensure the DRF of the program under strongly atomic semantics: $\mathsf{TDRF}(P)$. This contract lets the programmer check properties of a program assuming strongly atomic semantics ($[\![P]\!](\mathcal{H}_{\mathsf{atomic}})$) and get the guarantee that the properties hold when the program uses the actual TM implementation ($[\![P]\!](\mathcal{H})$). Theorem 10 follows from Lemma 7 and the next lemma, which is an adaptation of a result from [6].

▶ **Lemma 11.** *If $\mathcal{H}$ is a TM such that $\mathcal{H}|_{\mathsf{CDRF}} \sqsubseteq \mathcal{H}_{\mathsf{atomic}}$, then* $\forall P.\, \mathsf{CDRF}(P, \mathcal{H}) \implies [\![P]\!](\mathcal{H}) \preceq [\![P]\!](\mathcal{H}_{\mathsf{atomic}})$*.*

6 Proving Privatization-Safe Opacity

We now develop a method that reduces proving privatization-safe opacity ($\mathcal{H}|_{\mathsf{CDRF}} \sqsubseteq \mathcal{H}_{\mathsf{atomic}}$) to proving the ordinary opacity. The method builds on a *graph characterization* of opacity by Guerraoui and Kapalka [21], which was proposed for proving opacity of TMs that do not allow mixed transactional/non-transactional accesses to the same data. The characterization allows checking opacity of a history H by checking two properties: *consistency* of the history, denoted $\mathsf{cons}(H)$, and the acyclicity of a certain *opacity graph*, which we define in the following. Consistency is a basic well-formedness property of a history ensuring the following. If a transaction T in H reads a value of a register x and writes to it before, then T reads the latest value it writes. If T reads a value of x and does not write to it before, then it reads some value written non-transactionally or by a committed or commit-pending transaction (or the initial value, when everything else fails). Consistency also ensures that only the last write to x by a transaction is read from. We define consistency formally in [26, §D] and focus here on defining opacity graphs.

The vertexes in these graphs include transactions and non-transactional accesses in H. The intention of the vis predicate below is to mark those vertexes that have taken effect, including commit-pending transactions of this kind. The other components, intuitively, constrain the order in which the vertexes should appear in the atomic history.

▶ **Definition 12.** *The* opacity graph *of a history H is a tuple $G = (\mathcal{V}, \mathsf{vis}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}, \mathsf{PO}, \mathsf{CL})$, where:*

- *$\mathcal{V}$ is the set of graph vertexes, i.e., all transactions and non-transactional accesses from H (ranged over by ν).*
- *$\mathsf{vis} \subseteq \mathcal{V}$ is a* visibility *predicate, which holds of all non-transactional accesses and committed transactions and does not hold of all aborted and live transactions.*
- *$\mathsf{WR} : \mathsf{Reg} \to 2^{\mathcal{V}\times\mathcal{V}}$ specifies per-register* read-dependency *relations on vertexes, such that*
  - *For each read dependency $\nu \xrightarrow{\mathsf{WR}_x} \nu'$, we have that $\nu \neq \nu'$, ν contains $(_, _, \mathsf{write}(x, v))$, and ν' contains a request $(_, _, \mathsf{read}(x))$ and a matching response $(_, _, \mathsf{ret}(v))$.*
  - *Each vertex that reads x has at most one corresponding read dependency: $\forall \nu, \nu', \nu'', x.\, \nu \xrightarrow{\mathsf{WR}_x} \nu' \wedge \nu'' \xrightarrow{\mathsf{WR}_x} \nu \implies \nu = \nu''$.*
  - *Each vertex that is read from is visible: $\forall \nu, x.\, \nu \xrightarrow{\mathsf{WR}_x} _ \implies \mathsf{vis}(\nu)$.*

  Informally, $\nu \xrightarrow{\mathsf{WR}_x} \nu'$ means that ν' reads what ν wrote to x.
- *$\mathsf{WW} : \mathsf{Reg} \to 2^{\mathcal{V}\times\mathcal{V}}$ specifies per-register* write-dependency *relations, such that for each $x \in \mathsf{Reg}$, WW_x is an irreflexive total order on $\{\nu \in \mathcal{V} \mid \mathsf{vis}(\nu) \wedge (_, _, \mathsf{write}(x, _)) \in \nu\}$. Informally, $\nu \xrightarrow{\mathsf{WW}_x} \nu'$ means that ν' overwrites what ν wrote to x.*
- *$\mathsf{RW} \in \mathsf{Reg} \to 2^{\mathcal{V}\times\mathcal{V}}$ specifies per-register* anti-dependency *relations:*

$$\nu \xrightarrow{\mathsf{RW}_x} \nu' \iff \nu \neq \nu' \wedge ((\exists \nu''.\, \nu'' \xrightarrow{\mathsf{WW}_x} \nu' \wedge \nu'' \xrightarrow{\mathsf{WR}_x} \nu) \vee (\mathsf{vis}(\nu') \wedge (_, _, \mathsf{write}(x, _)) \in \nu' \wedge (_, _, \mathsf{ret}(x, v_{\mathsf{init}})) \in \nu)).$$

  Informally, $\nu \xrightarrow{\mathsf{RW}_x} \nu'$ means that ν' overwrites the write to x that ν previously read (the initial value of x is overwritten by any write to x).
- *$\mathsf{PO}, \mathsf{CL} \in 2^{\mathcal{V}\times\mathcal{V}}$ are the per-thread and client orders lifted to pairs of graph vertexes: e.g., $\nu \xrightarrow{\mathsf{PO}(H)} \nu' \iff \exists \alpha \in \nu, \alpha' \in \nu'.\, \alpha <_{\mathsf{po}(H)} \alpha'$.*

We let $\mathsf{Graph}(H)$ denote the set of all opacity graphs of H. We say that a graph G is *acyclic*, written $\mathsf{acyclic}(G)$, if its edges do not form a directed cycle. We also refer to histories resulting from topological sortings of vertexes in a graph G as its *linearizations* and denote their set by $\mathsf{lins}(G)$. The next lemma shows that we can check privatization-safe opacity of a history by checking its consistency and the acyclicity of its opacity graph, with any linearization of the graph yielding a matching atomic history.

▶ **Lemma 13.** $\forall H. (\mathsf{cons}(H) \wedge \exists G \in \mathsf{Graph}(H). \mathsf{acyclic}(G)) \implies \mathsf{lins}(G) \subseteq \mathcal{H}_{\mathsf{atomic}}$.

The lemma is proven analogously to Lemma 6.4 in [27, §B.2]. It implies the following theorem, which gives a criterion for the privatization-safe opacity of a TM $\mathcal{H}$.

▶ **Theorem 14.** $\mathcal{H} \sqsubseteq \mathcal{H}_{\mathsf{atomic}}$ *holds if* $\forall H \in \mathcal{H}. \mathsf{cons}(H) \wedge \exists G \in \mathsf{Graph}(H). \mathsf{acyclic}(G)$.

In comparison to the graph characterization of the ordinary opacity [21], ours is more complex: the graph includes non-transactional accesses and the acyclicity check has to take into account paths involving them. We now formulate lemmas that simplify reasoning about non-transactional operations: they allow proving the privatization-safe opacity of a TM using Theorem 14 with only small adjustments to a proof of its ordinary opacity using graph characterization. The latter characterization includes only transactions as nodes of the graph, but additionally considers paths including the lifting of the real-time order from §4 to transactions: for a history H, we let $\mathsf{RT}(H)$ be the relation between transactions in H such that $T <_{\mathsf{RT}(H)} T'$ iff for some $\alpha \in T$ and $\alpha' \in T$ we have $\alpha <_{\mathsf{rt}(H)} \alpha'$. We also let DEP denote any edge in a given graph G, and we let txDEP denote an edge between two transactions.

The following lemma exploits CDRF to show that, for every path between two transactions in an acyclic opacity graph, there is another path replacing edges involving non-transactional accesses by real-time order edges or transactional dependencies.

▶ **Lemma 15.** *Consider an acyclic opacity graph* $G = (\mathcal{V}, \mathsf{vis}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}, \mathsf{PO}, \mathsf{CL})$ *of a consistent CDRF history H. For any two transactions T and T', if* $T \xrightarrow{\mathsf{DEP}}{}^{*} T'$*, then* $T \xrightarrow{\mathsf{RT} \cup \mathsf{txDEP}}{}^{*} T'$.

The next lemma exploits CDRF to show that, for every path between a transaction and a non-transactional access in an acyclic opacity graph, there is another path where per-thread order is the only kind of an edge between transactions and non-transactional accesses.

▶ **Lemma 16.** *Consider an acyclic opacity graph* $G = (\mathcal{V}, \mathsf{vis}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}, \mathsf{PO}, \mathsf{CL})$ *of a CDRF history H. For any transaction T and non-transactional access n:*

- *if* $T \xrightarrow{\mathsf{DEP}}{}^{*} n$*, then there are T' and n' such that* $T \xrightarrow{\mathsf{RT} \cup \mathsf{txDEP}}{}^{*} T' \xrightarrow{\mathsf{PO}} n' \xrightarrow{\mathsf{CL}}{}^{*} n$*;*
- *if* $n \xrightarrow{\mathsf{DEP}}{}^{*} T$*, then there are T' and n' such that* $n \xrightarrow{\mathsf{CL}}{}^{*} n' \xrightarrow{\mathsf{PO}} T' \xrightarrow{\mathsf{RT} \cup \mathsf{txDEP}}{}^{*} T$.

Our method for proving the privatization-safe opacity of a TM (which we illustrate in §7) uses Lemmas 15 and 16 to reduce proving the acyclicity of an opacity graph to proving the absence of cycles in the projection of the graph to transactions, enriched with real-time order edges. The simplified acyclicity check is exactly the one required in the graph characterization of the ordinary opacity [21], allowing us to reuse existing proofs.

In the following we prove Lemmas 15 and 16. We show the existence of the paths required in the lemmas by using CDRF to eliminate WR/WW/RW-dependencies between transactions and non-transactional accesses. Each of the dependencies to be eliminated corresponds to a conflict in a matching atomic history, which CDRF guarantees to relate by happens-before. The next lemma exploits this observation.

▶ **Lemma 17.** *Consider an acyclic opacity graph $G = (\mathcal{V}, \mathsf{vis}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}, \mathsf{PO}, \mathsf{CL})$ of a consistent CDRF history H. For any transaction T and non-transactional access n:*
1. *if $T \xrightarrow{\mathsf{DEP}} n$, then there are T' and n' such that $T \xrightarrow{\mathsf{DEP}}{}^* T' \xrightarrow{\mathsf{PO}} n' \xrightarrow{\mathsf{CL}}{}^* n$;*
2. *if $n \xrightarrow{\mathsf{DEP}} T$, then there are T' and n' such that $n \xrightarrow{\mathsf{CL}}{}^* n' \xrightarrow{\mathsf{PO}} T' \xrightarrow{\mathsf{DEP}}{}^* T$.*

For example, consider an execution of the program in Figure 1 where T_2 reads `false` from `priv` and writes to `x` before n does. The corresponding acyclic graph contains both $T_2 \xrightarrow{\mathsf{WW_x}} n$ and $T_2 \xrightarrow{\mathsf{RW_{priv}}} T_1 \xrightarrow{\mathsf{PO}} n$. To prove Lemma 17, we lift $<_{\mathsf{po}(H)}$, $<_{\mathsf{ef}(H)}$, $<_{\mathsf{cl}(H)}$ and $<_{\mathsf{hb}(H)}$ from Definition 2 to vertexes of the graph as expected, writing $<_{\mathsf{PO}(H)}$, $<_{\mathsf{EF}(H)}$, $<_{\mathsf{CL}(H)}$ and $<_{\mathsf{HB}(H)}$ for the resulting relations. We also write $\leq$ for their reflexive closure. We rely on the following easy result (proved in [26, §D]).

▶ **Proposition 18.** *In a TDRF history H, for any T and n we have:*
- *if $T <_{\mathsf{HB}(H)} n$, then there are T' and n' such that $T \leq_{\mathsf{EF}(H)} T' <_{\mathsf{PO}(H)} n' \leq_{\mathsf{CL}(H)} n$;*
- *if $n <_{\mathsf{HB}(H)} T$, then there are T' and n' such that $n \leq_{\mathsf{CL}(H)} n' <_{\mathsf{PO}(H)} T' \leq_{\mathsf{EF}(H)} T$.*

Proof of Lemma 17. We only prove part 1, as part 2 can be proven analogously. Assume $T \xrightarrow{\mathsf{DEP}} n$. If $T \xrightarrow{\mathsf{PO}} n$, then $T \xrightarrow{\mathsf{DEP}}{}^* T \xrightarrow{\mathsf{PO}} n \xrightarrow{\mathsf{CL}}{}^* n$, which trivially concludes the proof. In the following, we consider the remaining case when $\neg(T \xrightarrow{\mathsf{PO}} n)$ and $T \xrightarrow{\mathsf{WR} \cup \mathsf{RW} \cup \mathsf{WW}} n$, so that T and n contain conflicting actions. Let $\mathcal{A}$ denote the following set of pairs (T', n') of a transaction and a non-transactional access:

$$\mathcal{A} \triangleq \{(T', n') \mid \exists L \in \mathsf{lins}(G).\, T \leq_{\mathsf{EF}(L)} T' <_{\mathsf{PO}(L)} n' \leq_{\mathsf{CL}(L)} n\}.$$

By Definition 12, for any $(T', n') \in \mathcal{A}$ we have $T' \xrightarrow{\mathsf{PO}} n' \xrightarrow{\mathsf{CL}}{}^* n$. It suffices to show that there is T' such that $T \xrightarrow{\mathsf{DEP}}{}^* T'$ and $(T', _) \in \mathcal{A}$. Proceeding by contradiction, let us assume that this is not the case: for every $(T', _) \in \mathcal{A}$, there is no edge $T \xrightarrow{\mathsf{DEP}}{}^* T'$ in G. Then extending the graph with edges $\{T' \xrightarrow{\mathsf{DEP}} T \mid (T', _) \in \mathcal{A}\}$ will not introduce a cycle. Hence, there exists a linearization $L \in \mathsf{lins}(G)$ in which every $(T', _) \in \mathcal{A}$ occurs before T: $\forall T'.\, (T', _) \in \mathcal{A} \implies T' <_{\mathsf{EF}(L)} T$.

Since, the history H is consistent and has an acyclic opacity graph G, by Lemma 13 we get $L \in \mathsf{lins}(G) \subseteq \mathcal{H}_{\mathsf{atomic}}$. Since H is CDRF, the conflicting pair T and n are ordered by $\mathsf{HB}(L)$. Moreover, since T occurs before n in L and $\mathsf{HB}(L)$ is consistent with the execution order of L, we have $T <_{\mathsf{HB}(L)} n$. From this by Proposition 18, for some T'' and n'' we have $T \leq_{\mathsf{EF}(L)} T'' <_{\mathsf{PO}(L)} n'' \leq_{\mathsf{CL}(L)} n$. Hence, $T \leq_{\mathsf{EF}(L)} T''$ and $(T'', n'') \in \mathcal{A}$. But by the construction of L we have $T'' <_{\mathsf{EF}(L)} T$, which contradicts the definition of ef as a total order on transactions. This contradiction demonstrates the required. ◀

The following result leverages Lemma 17 to show that, for every path between two transactions in an acyclic opacity graph, there is another path replacing some edges involving non-transactional accesses by real-time order edges or transactional dependencies.

▶ **Lemma 19.** *Consider an acyclic opacity graph $G = (\mathcal{V}, \mathsf{vis}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}, \mathsf{PO}, \mathsf{CL})$ of a consistent CDRF history H. For any two transactions T and T', if $T \xrightarrow{\mathsf{DEP}}{}^+ T'$, then there are two transactions T_1 and T_2 such that $T \xrightarrow{\mathsf{DEP}}{}^* T_1 \xrightarrow{\mathsf{txDEP} \cup \mathsf{RT}} T_2 \xrightarrow{\mathsf{DEP}}{}^* T'$.*

Proof. Assume $T \xrightarrow{\mathsf{DEP}}{}^+ T'$ and consider the corresponding path in the graph G. If there are no non-transactional accesses on this path, then $T \xrightarrow{\mathsf{txDEP}}{}^+ T'$, so the lemma holds trivially.

Assume now that there are non-transactional accesses on the path corresponding to $T \xrightarrow{\mathsf{DEP}}{}^+ T'$. Let n and n' be the first and the last such accesses respectively, and also let T_1' (T_2') be the transaction immediately preceding n (following n') on the path. Since G

is acyclic and CL relates every pair of non-transactional accesses, we must have $n \xrightarrow{\mathsf{CL}}{}^* n'$. Then $T \xrightarrow{\mathsf{DEP}}{}^* T_1' \xrightarrow{\mathsf{DEP}} n \xrightarrow{\mathsf{CL}}{}^* n' \xrightarrow{\mathsf{DEP}} T_2' \xrightarrow{\mathsf{DEP}}{}^* T'$. Applying Lemma 17(1) to $T_1' \xrightarrow{\mathsf{DEP}} n$ and Lemma 17(2) to $n' \xrightarrow{\mathsf{DEP}} T_2'$, we get that there are T_1, n_1, T_2 and n_2 such that:

$$T \xrightarrow{\mathsf{DEP}}{}^* T_1' \xrightarrow{\mathsf{DEP}}{}^* T_1 \xrightarrow{\mathsf{PO}} n_1 \xrightarrow{\mathsf{CL}}{}^* n \xrightarrow{\mathsf{CL}}{}^* n' \xrightarrow{\mathsf{CL}} n_2 \xrightarrow{\mathsf{PO}} T_2 \xrightarrow{\mathsf{DEP}}{}^* T_2' \xrightarrow{\mathsf{DEP}}{}^* T'.$$

Then $T \xrightarrow{\mathsf{DEP}}{}^* T_1 \xrightarrow{\mathsf{PO}} n_1 \xrightarrow{\mathsf{CL}}{}^* n_2 \xrightarrow{\mathsf{PO}} T_2 \xrightarrow{\mathsf{DEP}}{}^* T'$. By Definition 12 of PO and CL, T_1 ends before T_2 starts, so that $T_1 \xrightarrow{\mathsf{RT}} T_2$. Then $T \xrightarrow{\mathsf{DEP}}{}^* T_1 \xrightarrow{\mathsf{RT}} T_2 \xrightarrow{\mathsf{DEP}}{}^* T'$, as required. ◀

Proof of Lemma 15. To prove the lemma, we iteratively construct a path in G demonstrating that $T \xrightarrow{\mathsf{RT} \cup \mathsf{txDEP}}{}^* T'$. At the k-th iteration we construct a sequence π_k of transactions $T_0, T_0', T_1, T_1', \ldots, T_k, T_k' \in \mathcal{V}$ such that:

- $T_0 = T$, $T_k' = T'$, and
- $T_0 \xrightarrow{\mathsf{DEP}}{}^* T_0' \xrightarrow{\mathsf{RT} \cup \mathsf{txDEP}} T_1 \xrightarrow{\mathsf{DEP}}{}^* T_1' \xrightarrow{\mathsf{RT} \cup \mathsf{txDEP}} \ldots \xrightarrow{\mathsf{RT} \cup \mathsf{txDEP}} T_k \xrightarrow{\mathsf{DEP}}{}^* T_k'$.

We start the construction with a sequence $\pi_0 = T, T'$, which satisfies the above conditions because $T \xrightarrow{\mathsf{DEP}}{}^* T'$. We stop the construction once we get a sequence π_k such that $T_i = T_i'$ for each $i = 0..k$: in this case the sequence yields a path of the required form. Otherwise, we construct π_{k+1} from π_k as follows. We choose any two transactions T_i and T_i' in π_k such that $T_i \neq T_i'$ and, hence, $T_i \xrightarrow{\mathsf{DEP}}{}^+ T_i'$. By Lemma 19, there are T_i'' and T_i''' such that $T_i \xrightarrow{\mathsf{DEP}}{}^* T_i'' \xrightarrow{\mathsf{txDEP} \cup \mathsf{RT}} T_i''' \xrightarrow{\mathsf{DEP}}{}^* T_i'$. Then we let $\pi_{k+1} = T_0, T_0', \ldots, T_i, T_i'', T_i''', T_i', \ldots, T_k, T_k'$.

Since G is acyclic, in any π_k the only transactions that can coincide are some consecutive T_i and T_i'. Thus, π_k contains at least $k+1$ distinct transactions. But then our transformation has to stop after at most n steps, where n is the number of transactions in G. ◀

Proof of Lemma 16. We only prove part 1, as part 2 can be proven analogously. Assume $T \xrightarrow{\mathsf{DEP}}{}^* n$. Then there are T'' and n'' such that $T \xrightarrow{\mathsf{DEP}}{}^* T'' \xrightarrow{\mathsf{DEP}} n'' \xrightarrow{\mathsf{DEP}}{}^* n$. By Lemma 17, there are T' and n' such that $T \xrightarrow{\mathsf{DEP}}{}^* T'' \xrightarrow{\mathsf{DEP}}{}^* T' \xrightarrow{\mathsf{PO}} n' \xrightarrow{\mathsf{CL}}{}^* n'' \xrightarrow{\mathsf{CL}}{}^* n$. Then $T \xrightarrow{\mathsf{DEP}}{}^* T' \xrightarrow{\mathsf{PO}} n' \xrightarrow{\mathsf{CL}}{}^* n$. By Lemma 15, $T \xrightarrow{\mathsf{RT} \cup \mathsf{txDEP}}{}^* T'$, implying the required. ◀

As we show in [26, §D], the observations in the proofs of the Lemmas 15 and 16 additionally let us establish the following interesting theorem, giving an equivalent formulation of CDRF in terms of dependencies between transactions.

▶ **Theorem 20.** *Given a consistent history H, $\mathsf{CDRF}(H)$ holds if and only if in each acyclic opacity graph $G = (\mathcal{V}, \mathsf{vis}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}, \mathsf{PO}, \mathsf{CL}) \in \mathsf{Graph}(H)$ there is a path over edges from $\mathsf{PO} \cup \mathsf{CL} \cup \mathsf{txDEP} \cup \mathsf{RT}(H)$ between every pair of vertexes containing conflicting actions.*

7 Case Study: FencedTL2

In this section we illustrate how Lemmas 15 and 16 enable simple proofs of privatization-safe opacity using an example of a privatization-safe version of TL2 [14]. We give only the key parts of the proof and defer details to [26, §E]. There we also give a proof of privatization-safe opacity of a TM based on two-phase locking [21], which is privatization-safe.

As we noted in §1, the TL2 algorithm by itself is not privatization-safe. The reason is that TL2 executes transactions optimistically, buffering their writes, and flushes them to memory only on commit. Thus, in the example in Figure 1, it is possible for the transaction T_1 to privatize `x` and for n to modify it after T_2 started committing, but before its write to `x` reached the memory, so that T_2's write subsequently overwrites n's write and violates the

postcondition. We can make TL2 privatization-safe by modifying its implementation so that it executes a *transactional fence* [31, 35] at the end of every transaction, an implementation we call *FencedTL2*. The fence has a semantics similar to *Read-Copy-Update (RCU)* [32]: it blocks until all the concurrent transactions that were active when the fence was invoked complete, by either committing or aborting. For instance, in the example in Figure 1 executing a transactional fence after T_1 would block the thread until T_2 commits or aborts, thus ensuring that n is not overwritten by T_2's buffered write. The above way of making a TM privatization-safe is used in the GCC compiler [18] (albeit with TinySTM [16] instead of TL2) and has been experimentally evaluated in [36, 37].

To prove privatization-safe opacity of FencedTL2, for every one of its executions we inductively construct an opacity graph (with added real-time order edges) that matches its history. This is done with the help of the following *graph updates*, which specify how and when in the execution to extend the graph:

- At the start of a transaction T, a graph update $\textsc{txinit}(T)$ adds a new vertex T and extends the real-time order with edges $T' \xrightarrow{\mathsf{RT}} T$ for every completed transaction T'.
- At the end of a read operation of a transaction T reading from an object x, a graph update $\textsc{txread}(T, x)$ adds a read dependency $\nu \xrightarrow{\mathsf{WR}_x} T$, where ν is the vertex that wrote the value returned by the read.
- During the commit of a transaction T, TL2 validates the consistency of T's read-set before flushing T's write-set into memory. At the last step of the validation, a graph update $\textsc{txwrite}(T, x)$ adds a write dependency $\nu \xrightarrow{\mathsf{WW}_x} T$ for every object x in the write-set of T, where ν is the vertex that wrote the previous value of x.
- Upon each non-transactional write n to an object x, a graph update $\textsc{ntxwrite}(n, x)$ adds a new vertex n and a write-dependency $\nu \xrightarrow{\mathsf{WW}_x} n$, where ν is the vertex that wrote the previous value of x.
- Upon each non-transactional read n from an object x, a graph update $\textsc{ntxread}(n, x)$ adds a new vertex n and a read dependency $\nu \xrightarrow{\mathsf{WR}_x} n$, where ν is the vertex that wrote the value returned by n.

The updates also add anti-dependencies of the form $_ \xrightarrow{\mathsf{RW}} T$ induced by new read- and write-dependencies.

At each step of the graph construction we prove that the graph remains acyclic. Then Theorem 14 guarantees that the history of the execution is opaque. We use Lemmas 15 and 16 to reduce the task of proving the graph acyclicity to proving the absence of cycles involving transactions only. To discharge the latter proof obligation, we reuse our previous proof of opacity of TL2 [27], also done via the graph characterization. This proof establishes the following invariant over pairs (H, G) of a history H and a graph G:

- INV_1: H is a consistent history and the relation $\mathsf{txDEP} \cup \mathsf{RT}$ is acyclic.

To enable the reduction from privatization-safe to ordinary opacity, we prove the following invariant, which states the guarantee provided by fences in FencedTL2:

- INV_2: For every uncompleted transaction T and a transaction T', $T \xrightarrow{\mathsf{txDEP}}{}^{*}\, T' \xrightarrow{\mathsf{PO}} _$ does not hold.

An informal justification of INV_2 is as follows. By construction of the graph it is possible to establish that T' can depend on an uncompleted transaction T only when they execute concurrently. In this case, the fence of T' will wait for T to commit or abort, and until then there cannot be any transactions or non-transactional accesses in the thread of T' later in the per-thread order. By Theorem 14, privatization-safe opacity of FencedTL2 follows from

▶ **Theorem 21.** $\forall H \in \text{FencedTL2}.\, \mathsf{CDRF}(H) \implies \exists G.\, (H, G) \in \mathsf{INV}_1 \wedge \mathsf{INV}_2 \wedge \mathsf{acyclic}(G)$.

We prove Theorem 21 by induction on the length of the TM execution inducing H, constructing G as described above and showing that it remains acyclic after each update with the aid of the two invariants. Due to space constraints, we only explain how we prove acyclicity in the case of a graph update TXWRITE, which illustrates the use of Lemmas 15 and 16.

▶ **Lemma 22.** *Let (H', G') be the result of performing an update* TXWRITE(T, x) *on (H, G). Assume that $(H, G), (H', G') \in \mathsf{INV}_1 \wedge \mathsf{INV}_2$ and G is acyclic. Then G' is acyclic too.*

Proof. By contrapositive: we assume that G' contains a simple cycle and show that G' violates either INV_1 or INV_2. The graph update adds an edge of the form $_ \xrightarrow{\mathsf{WW}_x} T$ and the derived edges of the form $_ \xrightarrow{\mathsf{RW}_x} T$. Since both kinds of edges end in the same vertex T, they cannot occur in the same simple cycle. Hence, we can consider them separately.

Consider a simple cycle involving a new edge $\nu \xrightarrow{\mathsf{DEP}} T$ for some vertex ν. By our assumption, there must be a reverse path $T \xrightarrow{\mathsf{DEP}}{}^{*} \nu$ in G. Let us first consider the case when ν is a transaction T'. Since G is acyclic and H is consistent and CDRF, by Lemma 15 the path $T \xrightarrow{\mathsf{DEP}}{}^{*} T'$ can be reduced to $T \xrightarrow{\mathsf{RT} \cup \mathsf{txDEP}}{}^{*} T'$. Since G' only extends G, the same path is present in G' too. Then $T' \xrightarrow{\mathsf{txDEP}} T \xrightarrow{\mathsf{RT} \cup \mathsf{txDEP}}{}^{*} T'$ is a cycle over transactions in G', which contradicts $(H', G') \in \mathsf{INV}_1$. We now consider the case when ν is a non-transactional access n. Since G is acyclic and H is consistent and CDRF, by Lemma 16 there exist T' and n' such that $T \xrightarrow{\mathsf{txDEP}}{}^{*} T' \xrightarrow{\mathsf{PO}} n' \xrightarrow{\mathsf{CL}}{}^{*} n$ holds in G. Note that T is an uncompleted transaction, since it currently performs a graph update. Therefore, $T \xrightarrow{\mathsf{txDEP}}{}^{*} T' \xrightarrow{\mathsf{PO}} n' \xrightarrow{\mathsf{CL}}{}^{*} n$ is a contradiction to $(H, G) \in \mathsf{INV}_2$. ◀

8 The Cost of Privatization-Safety

We now present a result about the inherent cost of privatization-safety, by which we mean guaranteeing strongly atomic semantics to TDRF programs. In addition to TM histories, we consider the prefix-closed set of all TM *executions* $\mathcal{X}$, ranged over by φ. Unlike histories, they include internal TM actions that only occur in transactions and are not a part of the TM interface. One type of an internal action are *write-backs* of the form $(a, t, \mathsf{wb}(x, v))$, where $a \in \mathsf{ActionId}$, $t \in \mathsf{ThreadID}$, $x \in \mathsf{Reg}$, and $v \in \mathbb{Z}$. A write-back denotes a transaction of a thread t writing a value v to a register x. We assume that a TM implementation is represented by a pair $(\mathcal{H}, \mathcal{X})$ of a set of histories and a set of executions producing them.

▶ **Definition 23.** *A TM system $(\mathcal{H}, \mathcal{X})$ is* progressive *when for any $\varphi \in \mathcal{X}$ with at most one uncompleted transaction T, if the last interface action by T in φ is a request α, there exists a sequence of internal TM actions φ' by T and a response α' matching α such that $\varphi\varphi'\alpha' \in \mathcal{X}$.*

▶ **Definition 24.** *A TM system $(\mathcal{H}, \mathcal{X})$ has* invisible reads *when for any $\varphi\varphi' \in \mathcal{X}$ such that φ contains at most one uncompleted transaction T and φ' is a sequence of actions corresponding to another uncompleted transaction T' only conflicting with reads by T, if the last interface action by T' is a request α, there exists a sequence of internal TM actions φ'' by T' and a response $\alpha' \neq (_, _, \mathsf{aborted})$ matching α such that $\varphi\varphi'\varphi''\alpha' \in \mathcal{X}$.*

Our progressiveness property is analogous to obstruction-freedom [22], requiring a transaction to complete when running solo. Our invisible reads property can be ensured when the TM only writes to thread-local memory upon reading [21]. The FencedTL2 TM from §7 is privatization-safe and has invisible reads, but is not progressive due to its use of fences. As the following theorem shows, this is not accidental.

▶ **Theorem 25.** *A TM system that guarantees strongly atomic semantics to TDRF programs cannot both be progressive and have invisible reads.*

We rely on the following proposition, proved in [26, §G].

▶ **Proposition 26.** *Consider a TM system that guarantees strongly atomic semantics to TDRF programs. If φ is a TM execution of a single atomic block where the latter commits, and $(_, _, \mathsf{write}(x, v))$ is its last write request to x, then φ also contains a write-back $(_, _, \mathsf{wb}(x, v))$, and all write-backs to x occur in φ after the first write request to x.*

Proof of Theorem 25. The proof is by contradiction. Assume there exists a progressive TM $(\mathcal{H}, \mathcal{X})$ with invisible reads that guarantees strong atomicity to every TDRF program P, so that $[\![P]\!](\mathcal{H}) \preceq [\![P]\!](\mathcal{H}_{\mathsf{atomic}})$. We choose a particular TDRF program P and construct a counterexample trace from $[\![P]\!](\mathcal{H})$ that does not have a matching trace in $[\![P]\!](\mathcal{H}_{\mathsf{atomic}})$. Namely, we consider the following program P, similar to the one in Figure 1:

$$\{\,\texttt{priv} = \texttt{false} \wedge \texttt{x} = 0\,\}$$

```
l1 = atomic {                 ||  atomic {
  priv = true; } // T1        ||   if (!priv)
if (l1 == committed)          ||     x = 42;
  l2 = x; // n                ||  } // T2
```

We first consider a single-threaded program executing the atomic block in the right-hand-side thread t_2 of P. The TM always allows the program to execute requests (§2), and the invisible reads property ensures that the TM responds to them without aborting. Therefore, there is an execution $\varphi_2^0 \in \mathcal{X}$ consisting only of actions of the atomic block of t_2 in P ending with a commit-response. By Proposition 26, the execution of φ_2^0 contains a write-back $(_, t_2, \mathsf{wb}(\mathtt{x}, 42))$. Let φ_2 be the prefix of φ_2^0 until the first write-back $w = (_, t_2, \mathsf{wb}(\mathtt{x}, 42))$. By Proposition 26, φ_2 contains a write request to $\mathtt{x}$ and, therefore, a preceding response $(_, t_2, \mathsf{ret}(\mathtt{false}))$ to a read from $\mathtt{priv}$. The set of TM executions is prefix-closed, so $\varphi_2\, w \in \mathcal{X}$.

Note that φ_2 corresponds to a (partial) trace of P. We now let P continue φ_2 by executing the atomic block of the left-hand-side thread t_1. The TM always allows t_1 to execute requests (§2), and the invisible reads property ensures that the TM responds to them without aborting, as they only conflict with t_2's read from $\mathtt{priv}$ in φ_2. We thus obtain a sequence of actions φ_1 corresponding to a committed transaction T_1 such that $\varphi_2 \varphi_1 \in \mathcal{X}$. We can then execute $n = (_, t_1, \mathsf{read}(\mathtt{x}))(_, t_1, \mathsf{ret}(0))$, which returns the initial value of $\mathtt{x}$ as there has not been any write-back to $\mathtt{x}$ yet. We thereby obtain an execution $\varphi_2 \varphi_1 n \in \mathcal{X}$ in which thread t_1 of P has executed to completion.

We now let P resume executing the atomic block of thread t_2. Since the TM is progressive, the execution $\varphi_2 \varphi_1 n$ can be extended to an execution $\varphi = \varphi_2 \varphi_1 n \varphi_2' \in \mathcal{X}$ where the atomic block is completed, yielding a transaction T_2. We first consider the case when T_2 commits in φ. The execution φ corresponds to a trace $\tau \in [\![P]\!](\mathcal{H})$. Since $[\![P]\!](\mathcal{H}) \preceq [\![P]\!](\mathcal{H}_{\mathsf{atomic}})$, there exists a trace $\tau' \in [\![P]\!](\mathcal{H}_{\mathsf{atomic}})$ matching τ. Above we established that φ_2 reads $\mathtt{false}$ from $\mathtt{priv}$ and, hence, so does T_2. To justify reading this value in τ', T_2 must commit in this trace before T_1 starts and, therefore, before n starts too. Hence, n must observe T_2's write to $\mathtt{x}$ in τ', even though it observes the initial value in τ. Then τ' cannot match τ, and this contradiction concludes the proof.

We now consider the case when T_2 aborts in φ. Above we established that $\varphi_2^0 = \varphi_2\, w\, _ \in \mathcal{X}$, so that $\varphi_2\, w \in \mathcal{X}$. Since the TM executes write-backs as atomic writes, if a transaction is interrupted when a write-back w is pending, it proceeds with w once its execution resumes. Hence, it must be the case that φ_2' takes the form of $w \varphi_2''$, so that $\varphi = \varphi_2 \varphi_1 n w \varphi_2''$. Since

the TM does not impose restrictions on the placement of the non-transactional accesses (§2), it must also allow an execution $\varphi_2 \varphi_1 w n' \varphi_2'' \in \mathcal{X}$, where $n' = (_, t_1, \mathsf{read}(\mathtt{x}))(_, t_1, \mathsf{ret}(42))$ returns the value written by w. This execution corresponds to a trace $\tau \in [\![P]\!](\mathcal{H})$. Since $[\![P]\!](\mathcal{H}) \preceq [\![P]\!](\mathcal{H}_{\mathsf{atomic}})$, there exists a trace $\tau' \in [\![P]\!](\mathcal{H}_{\mathsf{atomic}})$ matching τ. In this trace n' reads 42 written by an aborted transaction T_2, which cannot happen under $\mathcal{H}_{\mathsf{atomic}}$. Hence, $\tau' \notin [\![P]\!](\mathcal{H}_{\mathsf{atomic}})$, and this contradiction concludes the proof. ◀

9 Related Work and Discussion

We have previously proposed a notion of DRF for privatization-unsafe TMs and a corresponding variant of opacity that ensure the Fundamental Property [27]. This work considered a more low-level programming model, which required inserting fences after some of the transactions for a program to be DRF. The resulting DRF notion was thus more involved than TDRF. Showing that the simpler TDRF is enough for privatization-safe TMs required us to address new technical challenges, such as the need to generalize TDRF to concurrent histories (to formulate privatization-safe opacity, §4) and to prove the delicate path reduction lemmas linking TDRF with properties of opacity graphs (§6). Furthermore, unlike [27], our results are also applicable to TMs that achieve privatization-safety by means other than fences, such as a lock-based TM we handle in [26, §F]. Our results also suggest a strengthening of those in [27]; we defer the details to [26, §H].

The notion of TDRF we use is a variant of the one proposed by Dalessandro et al. [11]. They also suggested that the notion should satisfy the Fundamental Property, but with strict serializability as the required condition on the TM. As we argued in §4, this condition is too strong, as it does not allow the proofs of TM correctness to benefit from the DRF of programs using it. In this paper we justify the appropriateness of TDRF by proposing a matching TM correctness condition that enables proofs of common TMs and proving the Fundamental Property for it. This also requires us to generalize TDRF to concurrent histories.

In this paper we assumed sequential consistency as a baseline non-transactional memory model. However, transactions are being integrated into languages, such as C++, that have weaker memory models [1]. Transactional sequential consistency, which we use as our strongly atomic semantics, is equivalent to that prescribed by the C++ memory model without relaxed transactions or non-SC atomics [9], and our definition of a data race is given in the axiomatic style used in the C++ memory model [2]. Hence, we believe that in the future our results can be generalized to the wider C++ model, in particular, by weakening the client order in Definition 2 to account for non-SC non-transactional accesses.

Abadi et al. also proposed disciplines for privatization with a formal justification of their safety [3, 4]. However, these disciplines are more restrictive than ours: they either prohibit mixing transactional and non-transactional accesses to the same register [4] or require explicit commands to privatize and publish an object [3]. Such disciplines are particular ways of achieving the more general notion of TDRF that we adopted.

Attiya and Hillel [7] investigated the cost of privatization in progressive TMs. Unlike us, they considered support for privatization to be part of TM interface and did not rely on a formal notion of privatization-safety. They proved the impossibility of supporting privatization in eager TMs, and a lower bound on its implementation cost in lazy TMs. Our Theorem 25 unifies and strengthens their results, as it states the impossibility of providing privatization-safety for all progressive TMs with invisible reads. We also make the results more rigorous by linking them to a formal notion of privatization-safety based on TDRF.

References

1 ISO/IEC. Technical Specification for C++ Extensions for Transactional Memory, 19841:2015, 2015.
2 ISO/IEC. Programming Languages – C++, 14882:2017, 2017.
3 Martín Abadi, Andrew Birrell, Tim Harris, Johnson Hsieh, and Michael Isard. Implementation and Use of Transactional Memory with Dynamic Separation. In *International Conference on Compiler Construction (CC)*, pages 63–77, 2009.
4 Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of transactional memory and automatic mutual exclusion. *ACM Trans. Program. Lang. Syst.*, 33:2:1–2:50, 2011.
5 Martín Abadi, Tim Harris, and Katherine F. Moore. A Model of Dynamic Separation for Transactional Memory. In *International Conference on Concurrency Theory (CONCUR)*, pages 6–20, 2008. doi:10.1007/978-3-540-85361-9_5.
6 Hagit Attiya, Alexey Gotsman, Sandeep Hans, and Noam Rinetzky. A programming language perspective on transactional memory consistency. In *Symposium on Principles of Distributed Computing (PODC)*, pages 309–318, 2013. doi:10.1145/2484239.2484267.
7 Hagit Attiya and Eshcar Hillel. The Cost of Privatization in Software Transactional Memory. *IEEE Transactions on Computers*, 62(12):2531–2543, 2013.
8 Colin Blundell, E. Christopher Lewis, and Milo M. K. Martin. Subtleties of Transactional Memory Atomicity Semantics. *Computer Architecture Letters*, 5(2), 2006.
9 Nathan Chong, Tyler Sorensen, and John Wickerson. The semantics of transactions and weak memory in x86, Power, ARM, and C++. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 211–225, 2018.
10 Luke Dalessandro and Michael L. Scott. Strong Isolation is a Weak Idea. In *Workshop on Transactional Computing (TRANSACT)*, 2009.
11 Luke Dalessandro, Michael L. Scott, and Michael F. Spear. Transactions as the Foundation of a Memory Consistency Model. In *International Symposium on Distributed Computing (DISC)*, pages 20–34, 2010.
12 Luke Dalessandro, Michael F. Spear, and Michael L. Scott. NOrec: streamlining STM by abolishing ownership records. In *Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 67–78, 2010. doi:10.1145/1693453.1693464.
13 Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid Transactional Memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 336–346, 2006.
14 David Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In *International Symposium on Distributed Computing (DISC)*, pages 194–208, 2006. doi:10.1007/11864219_14.
15 David Dice and Nir Shavit. TLRW: return of the read-write lock. In *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 284–293, 2010. doi:10.1145/1810479.1810531.
16 P. Felber, C. Fetzer, P. Marlier, and T. Riegel. Time-Based Software Transactional Memory. *IEEE Transactions on Parallel and Distributed Systems*, 21(12):1793–1807, 2010.
17 Ivana Filipovic, Peter O'Hearn, Noam Rinetzky, and Hongseok Yang. Abstraction for concurrent objects. *Theoretical Computer Science*, 411(51-52):4379–4398, 2010.
18 Free Software Foundation. Transactional Memory in GCC. URL: http://gcc.gnu.org/wiki/TransactionalMemory.
19 Rachid Guerraoui, Thomas A. Henzinger, Michal Kapalka, and Vasu Singh. Transactions in the jungle. In *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 263–272, 2010.
20 Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 175–184, 2008.

21 Rachid Guerraoui and Michal Kapalka. *Principles of Transactional Memory*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2010. doi:10.2200/S00253ED1V01Y201009DCT004.

22 Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-Free Synchronization: Double-Ended Queues as an Example. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 522–529, 2003.

23 Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *International Symposium on Computer Architecture (ISCA)*, pages 289–300, 1993. doi:10.1145/165123.165164.

24 Intel Corporation. Intel architecture instruction set extensions programming reference. Chapter 8: Intel transactional synchronization extensions, 2012.

25 Gokcen Kestor, Osman S. Unsal, Adrián Cristal, and Serdar Tasiran. T-Rex: a dynamic race detection tool for C/C++ transactional memory applications. In *European Systems Conference (EuroSys)*, pages 20:1–20:12, 2014.

26 Artem Khyzha, Hagit Attiya, and Alexey Gotsman. Privatization-Safe Transactional Memories (Extended Version). *arXiv CoRR*, 1908.03179, 2019. arXiv:1908.03179.

27 Artem Khyzha, Hagit Attiya, Alexey Gotsman, and Noam Rinetzky. Safe privatization in transactional memory. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages pages 233–245, 2018. Extended version available at arXiv:1801.04249.

28 Sanjeev Kumar, Michael Chu, Christopher J Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 209–220, 2006.

29 H. Q. Le, G. L. Guthrie, D. E. Williams, M. M. Michael, B. G. Frey, W. J. Starke, C. May, R. Odaira, and T. Nakaike. Transactional memory support in the IBM POWER8 processor. *IBM Journal of Research and Development*, 59(1):8:1–8:14, 2015.

30 Mohsen Lesani, Victor Luchangco, and Mark Moir. Specifying Transactional Memories with Nontransactional Operations. In *Workshop on the Theory of Transactional Memory (WTTM)*, 2013.

31 Virendra J. Marathe, Michael F. Spear, and Michael L. Scott. Scalable Techniques for Transparent Privatization in Software Transactional Memory. In *International Conference on Parallel Processing (ICPP)*, pages 67–74, 2008.

32 Paul E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004.

33 Katherine F. Moore and Dan Grossman. High-level small-step operational semantics for transactions. In *Symposium on Principles of Programming Languages (POPL)*, pages 51–62, 2008. doi:10.1145/1328438.1328448.

34 Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

35 Michael F. Spear, Virendra J. Marathe, Luke Dalessandro, and Michael L. Scott. Privatization techniques for software transactional memory. In *Symposium on Principles of Distributed Computing (PODC)*, pages 338–339, 2007. Extended version appears as Technical Report 915, Computer Science Department, University of Rochester.

36 Richard M. Yoo, Yang Ni, Adam Welc, Bratin Saha, Ali-Reza Adl-Tabatabai, and Hsien-Hsin S. Lee. Kicking the tires of software transactional memory: why the going gets tough. In *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 265–274, 2008. doi:10.1145/1378533.1378582.

37 Tingzhe Zhou, Pantea Zardoshti, and Michael F. Spear. Practical Experience with Transactional Lock Elision. In *International Conference on Parallel Processing (ICPP)*, pages 81–90, 2017.

Low-Congestion Shortcut and Graph Parameters

Naoki Kitamura
Nagoya Institute of Technology, Japan
31514002@stn.nitech.ac.jp

Hirotaka Kitagawa
Nagoya Institute of Technology, Japan

Yota Otachi
Kumamoto University, Japan
otachi@cs.kumamoto-u.ac.jp

Taisuke Izumi
Nagoya Institute of Technology, Japan
t-izumi@nitech.ac.jp

Abstract

Distributed graph algorithms in the standard CONGEST model often exhibit the time-complexity lower bound of $\tilde{\Omega}(\sqrt{n} + D)$ rounds for many global problems, where n is the number of nodes and D is the diameter of the input graph. Since such a lower bound is derived from special "hard-core" instances, it does not necessarily apply to specific popular graph classes such as planar graphs. The concept of *low-congestion shortcuts* is initiated by Ghaffari and Haeupler [SODA2016] for addressing the design of CONGEST algorithms running fast in restricted network topologies. Specifically, given a specific graph class X, an f-round algorithm of constructing shortcuts of quality q for any instance in X results in $\tilde{O}(q + f)$-round algorithms of solving several fundamental graph problems such as minimum spanning tree and minimum cut, for X. The main interest on this line is to identify the graph classes allowing the shortcuts which are efficient in the sense of breaking $\tilde{O}(\sqrt{n} + D)$-round general lower bounds.

In this paper, we consider the relationship between the quality of low-congestion shortcuts and three major graph parameters, chordality, diameter, and clique-width. The main contribution of the paper is threefold: (1) We show an $O(1)$-round algorithm which constructs a low-congestion shortcut with quality $O(kD)$ for any k-chordal graph, and prove that the quality and running time of this construction is nearly optimal up to polylogarithmic factors. (2) We present two algorithms, each of which constructs a low-congestion shortcut with quality $\tilde{O}(n^{1/4})$ in $\tilde{O}(n^{1/4})$ rounds for graphs of $D = 3$, and that with quality $\tilde{O}(n^{1/3})$ in $\tilde{O}(n^{1/3})$ rounds for graphs of $D = 4$ respectively. These results obviously deduce two MST algorithms running in $\tilde{O}(n^{1/4})$ and $\tilde{O}(n^{1/3})$ rounds for $D = 3$ and 4 respectively, which almost close the long-standing complexity gap of the MST construction in small-diameter graphs originally posed by Lotker et al. [Distributed Computing 2006]. (3) We show that bounding clique-width does not help the construction of good shortcuts by presenting a network topology of clique-width six where the construction of MST is as expensive as the general case.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases distributed graph algorithms, low-congestion shortcut, k-chordal graph, clique width, minimum spanning tree

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.25

Related Version `https://arxiv.org/abs/1908.09473`

Funding *Naoki Kitamura*: This work was supported by JSPS KAKENHI Grant Number JP19J22696, Japan.
Yota Otachi: This work was supported by JSPS KAKENHI Grant Numbers JP18H04091, JP18K11168, and JP18K11169, Japan.
Taisuke Izumi: This work was supported by JST SICORP Grant Number JPMJSC1606 and JSPS KAKENHI Grant Number JP19K11824, Japan.

© Naoki Kitamura, Hirotaka Kitagawa, Yota Otachi, and Taisuke Izumi;
licensed under Creative Commons License CC-BY

33rd International Symposium on Distributed Computing (DISC 2019).
Editor: Jukka Suomela; Article No. 25; pp. 25:1–25:17

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

1.1 Background

The *CONGEST* is one of the standard message-passing models in the development of distributed graph algorithms, especially for global problems such as shortest paths and minimum spanning tree. It is a round-based synchronous system where each link can transfer $O(\log n)$-bit information per one round (n is the number of nodes in the system). Since most of global distributed tasks as mentioned above inherently require each node to access the information far apart from itself, it is not possible to "localize" the communication assessed for solving those tasks. That is, the $\Omega(D)$-round complexity often becomes an *universal* lower bound applied to any network topology, where D is the diameter of the input topology. While D-round computation is sufficiently long to make some information reach all the nodes in the network, the constraint of limited bandwidth precludes the centralized solution that one node collects the information of whole network topology because it results in expensive $\Omega(n)$-round time complexity. The round complexity of CONGEST algorithms solving global tasks is typically represented in the form of $\tilde{O}(n^c + D)$ or $\tilde{O}(n^c D)$ for some constant $0 \leq c \leq 2$[1], and thus the main complexity-theoretic question is how much we can make c small (ideally $c = 0$, which matches the universal lower bound). Unfortunately, achieving such an universal bound is an impossible goal for many problems, e.g., minimum spanning tree (MST), shortest paths, minimum cut, and so on. They exhibit the lower bound of $\tilde{\Omega}(\sqrt{n} + D)$ rounds for general graphs.

Most of $\tilde{\Omega}(n^c + D)$-round lower bounds for some $c > 0$ are derived from special "hard-core" instances, and does not necessarily apply to popular graph classes such as planar graphs, which evokes the interest of developing efficient distributed graph algorithms for specific graph classes. In the last few years, the study along this line rapidly made progress, where the concepts of *partwise aggregation* and *low-congestion shortcuts* play an important role. In the partwise aggregation problem, all the nodes in the network is initially partitioned into a number of disjoint connected subgraphs, which we call a *part*. The goal of this problem is to perform a certain kind of distributed tasks independently within all the parts in parallel. The executable tasks cover several standard operations such as broadcast, convergecast, leader election, finding minimum, and so on. The *low-congestion shortcut* is a framework of solving the partwise aggregation problem, which is initiated by Ghaffari and Haeupler [11]. The key difficulty of the partwise aggregation problem appears when the diameter of a part is much larger than the diameter D of the original graph. Since the diameter can become $\Omega(n)$ in the worst case, the naive solution which performs the aggregation task only by in-part communication can cause the expensive $\Omega(n)$-round running time. A low-congestion shortcut is defined as the sets of links augmented to each part for accelerating the aggregation task there. Its efficiency is characterized by two quality parameters: The *dilation* is the maximum diameter of all the parts after the augmentation, and the *congestion* is the maximum edge congestion of all edges e, where the edge congestion of e is defined as the number of the parts augmenting e. In the application of low-congestion shortcuts, the performance of an algorithm typically relies on the sum of the dilation and congestion. Hence we simply call the value of dilation plus congestion the *quality* of the shortcut. It is known that any low-congestion shortcut with quality q and $O(f)$-round construction time yields an $\tilde{O}(f + q)$-round solution for the partwise aggregation problem, and $\tilde{O}(f + q)$-round partwise aggregation yields the efficient solutions for several fundamental graph problems. Precisely, the following meta-theorem holds.

[1] $\tilde{O}(\cdot)$ is a notation which ignores polylog(n) factors from $O(\cdot)$.

▶ **Theorem 1** (Ghaffari and Haeupler [11], Haeupler and Li [19]). *Let $\mathcal{G}$ be a graph class allowing the low-congestion shortcut with quality $O(q)$ that can be constructed in $O(f)$ rounds in the CONGEST model. Then there exist three algorithms solving (1) the MST problem in $\tilde{O}(f+q)$ rounds, (2) the $(1+\epsilon)$-approximate minimum cut problem in $\tilde{O}(f+q)$ rounds for any $\epsilon = \Omega(1)$, and (3) $O(n^{O(\log\log n)/\log\beta})$-approximate weighted single-source shortest path problem in $\tilde{\Omega}((f+q)\beta)$ rounds for any $\beta = \Omega(\mathrm{polylog}(n))$*[2]*.*

Conversely, if we get a time-complexity lower bound for any problem stated above, then it also applies to the partwise aggregation and low-congestion shortcuts (with respect to quality plus construction time). In fact, the $\tilde{O}(\sqrt{n}+D)$-round lower bound of shortcuts for general graphs is deduced from the lower bound of MST. On the other hand, the existence of efficient (in the sense of breaking the general lower bound) low-congestion shortcuts is known for several major graph classes, as well as its construction algorithms [11, 13, 14, 17, 18, 20].

1.2 Our Result

In this paper, we study the relationship between several major graph parameters and the quality of low-congestion shortcuts. Specifically, we focus on three parameters, that is, (1) chordality, (2) diameter, and (3) clique-width. The precise statement of our result is as follows:

- There is an $O(1)$-round algorithm which constructs a low-congestion shortcut with quality $O(kD)$ for any k-chordal graph. When $k = O(1)$, its quality matches the $\Omega(D)$-universal lower bound.
- For $k \leq D$ and $kD \leq \sqrt{n}$, there exists a k-chordal graph where the construction of MST requires $\tilde{\Omega}(kD)$ rounds. It implies that the quality plus construction time of our algorithm is nearly optimal up to polylogarithmic factors.
- There exists an algorithm of constructing a low-congestion shortcut with quality $\tilde{O}(n^{1/4})$ in $\tilde{O}(n^{1/4})$ rounds for any graph of diameter three. In addition, there exists an algorithm of constructing a low-congestion shortcut with quality $\tilde{O}(n^{1/3})$ in $\tilde{O}(n^{1/3})$ rounds for any graph of diameter four. These results almost close the long-standing complexity gap of the MST construction in graphs with small diameters, which is originally posed by Lotker et al. [24].
- We present a negative instance certifying that bounded clique-width does not help the construction of good-quality shortcuts. Precisely, we give an instance of clique-width six where the construction of MST is as expensive as the general case, i.e., $\tilde{\Omega}(\sqrt{n}+D)$ rounds.

Table 1 summarizes the state-of-the-art upper and lower bounds for low-congestion shortcuts. It should be noted that all the parameters considered in this paper is independent of the other parameters such that bounding it admits good shortcuts (e.g., treewidth and genus), and thus any result above is not a corollary of the past results.

For proving our upper bounds, we propose a new scheme of shortcut construction, called *1-hop extension*, where each node in a part only takes all the incident edges as the shortcut edges of its own part. Surprisingly, this very simple construction admits an optimal shortcut for any k-chordal graph. For graphs of diameter three or four, our algorithm is obtained by combining the 1-hop extension scheme with yet another algorithm of finding short low-congestion paths (i.e., paths of length one or two) connecting two moderately-large

[2] The statement of the weighted single-source shortest path problem is slightly simplified. See [19] for the details.

subgraphs. These algorithms are still simple but it is far from triviality to bound the quality of constructed shortcuts. The analytic part includes several (seemingly) new ideas and may be of independent interest.

Table 1 The quality bounds of Low-Congestion Shortcuts for Specific Graph Classes.

Graph Family	Quality	Construction	Lower bound
General	$\tilde{O}(\sqrt{n}+D)$ [22]	$\tilde{O}(\sqrt{n}+D)$ [22]	$\Omega(\sqrt{n}+D)$ [29]
Planar	$\tilde{O}(D)$ [11]	$\tilde{O}(D)$ [11]	$\tilde{\Omega}(D)$ [11]
Genus-g	$\tilde{O}(\sqrt{g}D)$ [18]	$\tilde{O}(\sqrt{g}D)$ [18]	$\tilde{\Omega}(\sqrt{g}D)$ [18]
Treewidth-k	$\tilde{O}(kD)$ [18]	$\tilde{O}(kD)$ [18]	$\Omega(kD)$ [18]
Clique-width-6	–	–	$\tilde{\Omega}(\sqrt{n}+D)$ **(this paper)**
Expander	$\tilde{O}\left(\tau 2^{O\left(\sqrt{\log n}\right)}\right)$ [14]*)	$\tilde{O}\left(\tau 2^{O\left(\sqrt{\log n}\right)}\right)$ [14]	–
k-Chordal	$O(kD)$ **(this paper)**	$O(1)$ **(this paper)**	$\tilde{\Omega}(kD)$ **(this paper)**
Excluded Minor	$\tilde{O}(D^2)$ [20]	$\tilde{O}(D^2)$ [20]	–
$D=3$	$\tilde{O}(n^{1/4})$ **(this paper)**	$\tilde{O}(n^{1/4})$ **(this paper)**	$\Omega(n^{1/4})$ [24, 30]
$D=4$	$\tilde{O}(n^{1/3})$ **(this paper)**	$\tilde{O}(n^{1/3})$ **(this paper)**	$\Omega(n^{1/3})$ [24, 30]
$5 \leq D \leq \log n$	–	–	$\tilde{\Omega}\left(n^{(D-2)/(2D-2)}\right)$ [30]

*) τ is the mixing time of the network graph G.

1.3 Related Work

The MST problem is one of the most fundamental problems in distributed graph algorithms. It is not only important by itself, but also has many applications for solving other distributed tasks (e.g., detecting connected components, minimum cut, and so on). Hence many researches have tackled the design of efficient MST algorithms in the CONGEST model so far [7, 8, 12, 15, 16, 21, 22, 27, 28]. The round-complexity lower bound of MST construction is also a central topic in distributed complexity theory [5, 6, 24, 25, 29, 30]. The inherent difficulty of MST construction is of solving the partwise aggregation (minimum) problem efficiently. This viewpoint is first identified by Ghaffari and Haeupler [11] explicitly, as well as an efficient algorithm for solving it in planar graphs. The concept of low-congestion shortcuts is newly invented there for encapsulating the difficulty of partwise aggregation. Recently, several follow-up papers are published to extend the applicability of low-congestion shortcuts, which break the known general lower bounds of several fundamental graph problems in several specific graph classes: This line includes bounded-genus graphs [11, 17], bounded-treewidth graphs [17], graphs with excluded minors [20], expander graphs [13, 14], and so on (see Table 1).

The application of low-congestion shortcuts is not limited only to MST. As stated in Theorem 1, it also admits efficient solutions for approximate minimum cut and single-source shortest path. A few algorithms recently proposed utilize low-congestion shortcuts as an important building block, e.g., the depth first search in planar graphs [19] and approximate treewidth (with decomposition) [23]. Haeupler et al. [16] shows a message-reduction scheme of shortcut-based algorithms, which drop the total number of messages exchanged by the algorithm into $\tilde{O}(m)$, where m is the number of links. On the negative side, it is known that the hardness of (approximate) diameter cannot be encapsulated by low-congestion shortcuts. Abboud et al. [1] shows a hard-core family of unweighted graphs with $O(\log n)$ treewidth where any diameter computation in the CONGEST model requires $\tilde{\Omega}(n)$ rounds. Since any

graph with $O(\log n)$ treewidth admits a low-congestion shortcut of quality $\tilde{O}(D)$, this result implies that it is not possible to compute the diameter of graphs efficiently by using only the property of low-congestion shortcuts.

While our results exhibit a tight upper bound for graphs of diameter three or four, a more generalized lower bound is known for small-diameter graphs. [30]. For any $\log n \geq D \geq 3$, it is proved that there exists a network topology which incurs the $\tilde{\Omega}\left(n^{(D-2)/(2D-2)}\right)$-round time complexity for any MST algorithm. In more restricted cases of $D = 1$ and $D = 2$, Jurdzinski et al. [21] and Lotker et al. [24] respectively show $O(1)$-round and $O(\log n)$-round MST algorithms.

1.4 Outline of the Paper

The paper is organized as follows: In Section 2, we introduce the formal definitions of the CONGEST model, partwise aggregation, and low-congestion shortcuts, and other miscellaneous terminologies and notations. In Section 3, we show the upper and lower bounds for shortcuts and MST in k-chordal graphs. In Section 4, we present our shortcut algorithms for graphs of diameter three or four. In Section 5, we prove the hardness result for bounded clique-width graphs. The paper is concluded in Section 6.

2 Preliminaries

2.1 CONGEST model

Throughout this paper, we denote by $[a, b]$ the set of integers at least a and at most b. A distributed system is represented by a simple undirected connected graph $G = (V, E)$, where V is the set of nodes and E is the set of edges. Let n and m be the numbers of nodes and edges respectively, and D be the diameter of G. Each node has an *ID* from $\mathbb{N}$ (which is represented with $O(\log n)$ bits). In the CONGEST model, the computation follows the round-based synchrony. In one round, each node sends messages to its neighbors, receives messages from its neighbors, and executes local computation. It is guaranteed that every message sent at a round is delivered to the destination within the same round. Each link can transfer $O(\log n)$-information (bidirectionally) per one round, and each node can inject different messages to its incident links. Each node has no prior knowledge on the network topology except for its neighbor's IDs. Given a graph H for which the node and link sets are not explicitly specified, we denote them by V_H and E_H respectively. Let $N(v)$ be the set of nodes that are adjacent to v, and $N^+(v) = N(v) \cup \{v\}$. We define $N(S) = \cup_{s \in S} N(s)$ and $N^+(S) = \cup_{s \in S} N^+(s)$ for any $S \subseteq V$. For two node subsets $X, Y \subseteq V$, we also define $E(X, Y) = \{(u, v) \in E \mid u \in X, v \in Y\}$. If X (resp. Y) is a singleton $X = \{w\}$, (resp. $Y = \{w\}$), we describe $E(X, Y)$ as $E(w, Y)$ (resp. $E(X, w)$). The *distance* (i.e., the number of edges in the shortest path) between two nodes u and v in G is denoted by $\text{dist}_G(u, v)$. Let S be a path in G. With a small abuse of notations, we often treat S as the sequence of nodes or edges representing the path, as the set of nodes or edges in the path, or the subgraph of G forming the path.

2.2 Partwise Aggregation

The *partwise aggregation* is a communication abstraction defined over a set $\mathcal{P} = \{P_1, P_2, \ldots, P_N\}$ of mutually-disjoint and connected subgraphs called *parts*, and provides simultaneous fast communication among the nodes in each P_i. It is formally defined as follows:

▶ **Definition 2** (Partwise Aggregation (PA)). *Let $\mathcal{P} = \{P_1, P_2, \ldots, P_N\}$ be the set of connected mutually-disjoint subgraphs of G, and each node $v \in V_{P_i}$ maintains variable b_v^i storing an input value $x_v^i \in X$. The output of the partwise aggregation problem is to assign $\oplus_{w \in P_i} x_w^i$ with b_v^i for any $v \in V_{P_i}$, where $\oplus$ is an arbitrary associative and commutative binary operation over X.*

The straightforward solution of the partwise aggregation problem is to perform the convergecast and broadcast in each part P_i independently. Specifically, we construct a BFS tree for each part P_i (after the selection of the root by any leader election algorithm). The time complexity is proportional to the diameter of each part P_i, which can be large ($\Omega(n)$ in the worst case) independently of the diameter of G.

2.3 (d, c)-Shortcut

As we stated in the introduction, the notion of low-congestion shortcuts is introduced for quickly solving the partwise aggregation problem (for some specific graph classes). The formal definition of (d, c)-shortcuts is given as follows.

▶ **Definition 3** (Ghaffari and Haeupler [11]). *Given a graph $G = (V, E)$ and a partition $\mathcal{P} = \{P_1, P_2, \ldots, P_N\}$ of G into node-disjoint and connected subgraphs, we define a (d, c)-shortcut of G and $\mathcal{P}$ as a set of subgraphs $\mathcal{H} = \{H_1, H_2, \ldots, H_N\}$ of G such that:*

1. *For each i, the diameter of $P_i + H_i$ is at most d (d-dilation).*
2. *For each edge $e \in E$, the number of subgraphs $P_i + H_i$ containing e is at most c (c-congestion).*

The values of d and c for a (d, c)-shortcut $\mathcal{H}$ is called the *dilation* and *congestion* of $\mathcal{H}$. As a general statement, a (d, c)-shortcut which is constructed in f rounds admits the solution of the partwise aggregation problem in $\tilde{O}(d + c + f)$ rounds [10, 11]. Since the parameter $d + c$ asymptotically affects the performance of the application, we call the value of $d + c$ the *quality* of (d, c)-shortcuts. A low-congestion shortcut with quality q is simply called a *q-shortcut.*

2.4 The framework of the Lower Bound

To prove the lower bound of MST, we introduce a simplified version of the framework by Das Sarma et al. [30]. In this framework, we consider the graph class $\mathcal{G}(n, b, l, c)$ that is defined below. A vertex set $X \subseteq V$ is called *connected* if the subgraph induced by X is connected.

▶ **Definition 4.** *For $n, b, c \geq 0$ and $l \geq 3$, the graph class $\mathcal{G}(n, b, l, c)$ is defined as the set of n-vertex graph $G = (V, E)$ satisfying the following conditions:*

- ***(C1)** The vertex set V is partitioned into ℓ disjoint vertex sets $\mathcal{X} = \{X_1, X_2, \ldots, X_\ell\}$ such that X_1 and X_ℓ are singletons (let $X_1 = \{s\}$ and $X_\ell = \{r\}$).*
- ***(C2)** The vertex set $V \setminus \{s, r\}$ is partitioned into b disjoint connected sets $\mathcal{Q} = \{Q_1, \ldots, Q_b\}$ such that $|E(X_1, Q_i)| \geq 1$ and $|E(X_l, Q_i)| \geq 1$ hold for any $1 \leq i \leq b$.*
- ***(C3)** Let $R_i = \bigcup_{i+1 \leq j \leq l} X_j$ and $L_i = \bigcup_{0 \leq j \leq l-1-i} X_j$. For $2 \leq i \leq l/2-1$, $|E(R_i, N(R_i) \setminus R_{i-1})| \leq c$ and $|E(L_i, N(L_i) \setminus L_{i-1})| \leq c$.*

Figure 1 shows the graph that is defined vertex partition $\mathcal{X}$ and $\mathcal{Q}$ for the hard-core instances presented in the original proof by Das Sarma et al. [30]. This graph belongs to $\mathcal{G}(O(lb), b, l, O(\log n))$. For class $\mathcal{G}(n, b, l, c)$, the following theorem holds, which is just a corollary of the result by Das Sarma et al. [30].

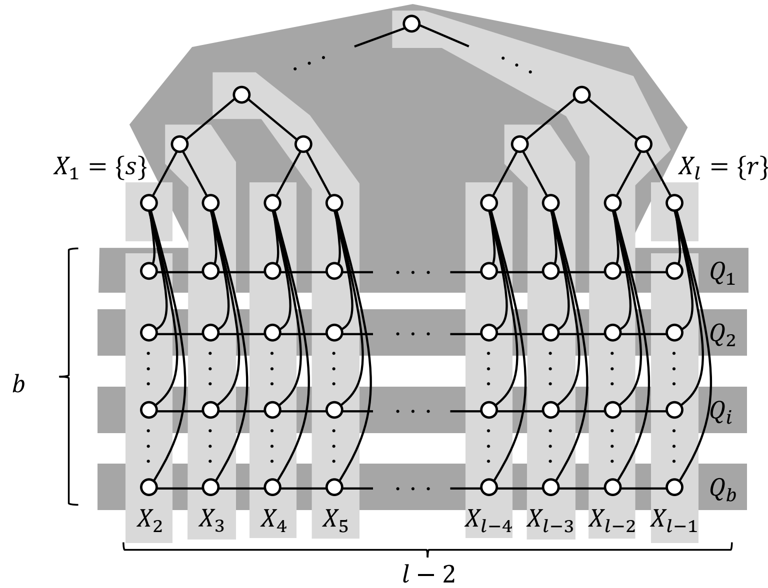


Figure 1 Example of $\mathcal{G}(O(lb), b, l, O(\log n))$.

▶ **Theorem 5** (Das Sarma et al. [30]). *For any graph $G \in \mathcal{G}(n, b, l, c)$ and any MST algorithm A, there exists an edge-weight function $w_{A,G} : E \to \mathbb{N}$ such that the execution of A in G requires $\tilde{\Omega}(\min\{b/c, l/2 - 1\})$ rounds. This bound holds with high probability even if A is a randomized algorithm.*

3 Low-Congestion Shortcut for k-Chordal Graphs

3.1 k-Chordal Graph

A graph G is *k-chordal* if and only if every cycle of length larger than k has a chord (equivalently, G contains no induced cycle of length larger than k). In particular, 3-chordal graphs are simply called *chordal* graphs, which is known to be much related to various intersection graph families such as interval graphs [9, 26]. Since k-chordal graphs can contain the clique of an arbitrary size for any $k \geq 3$, it is never a subclass of any minor-excluded graphs. Thus no known shortcut algorithm works correctly for k-chordal graphs. The main results of this section are the following two theorems:

▶ **Theorem 6.** *There is an $O(1)$-round algorithm which constructs a $O(kD)$-shortcut for any k-chordal graph.*

▶ **Theorem 7.** *For $k \leq D$ and $kD \leq \sqrt{n}$, there exists an unweighted k-chordal graph $G = (V, E)$ where for any MST algorithm A, there exists an edge-weight function $w_A : E \to \mathbb{N}$ such that the running time of A becomes $\tilde{\Omega}(kD)$ rounds.*

3.2 Proof of Theorem 6

We provide the proof of Theorem 6. The construction algorithm is very simple. It follows the *1-hop extension* scheme stated below:

For any $V_{P_i} \subseteq V$, node $v \in V_{P_i}$ adds each incident edge (v, u) to H_i, and informs u of the fact of $(v, u) \in H_i$.

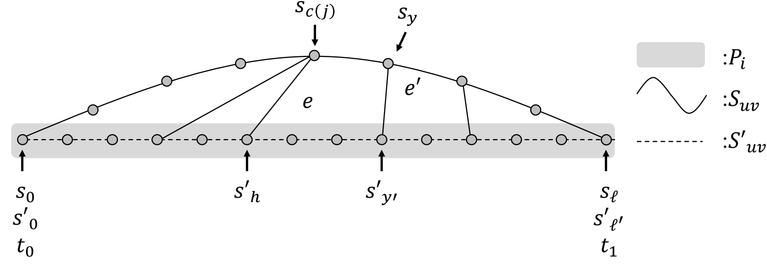


Figure 2 Proof of Lemma 8.

Obviously, this algorithm terminates in one round. Since each node belongs to one part, the congestion of each edge is at most two. Therefore, the technical challenge in proving Theorem 6 is to show that the diameter of $P_i + H_i$ is $O(kD)$ for any $i \in [1, N]$. In other words, the following lemma trivially deduces Theorem 6.

▶ **Lemma 8.** *Letting $G_i = P_i + H_i$, $\mathrm{dist}_{G_i}(u, v) \leq kD + 2$ holds for any $u, v \in V_{G_i}$.*

Proof. We show that $\mathrm{dist}_{G_i}(u, v) \leq kD$ holds for any $u, v \in V_{P_i}$. Since any node in $v \in V_{G_i} \setminus V_{P_i}$ is a neighbor of a node in V_{P_i}, it obviously follows the lemma.

Let A be the shortest path from u to v in G, and B be that in P_i. We define $T = (t_0, t_1, \ldots, t_{z-1})$ as the sequence of nodes in $A \cap B$ which are sorted in the order of A. By definition, $u = t_0$ and $v = t_{z-1}$ holds. The core of the proof is to show that $\mathrm{dist}_{G_i}(t_x, t_{x+1}) \leq k \cdot \mathrm{dist}_G(t_x, t_{x+1})$ for $0 \leq x \leq z - 1$. Summing up this inequality for all x, we obtain $\mathrm{dist}_{G_i}(t_0, t_{z-1}) \leq \sum_{1 \leq j \leq z} k\mathrm{dist}_G(t_{j-1}, t_j) = kD$. By symmetry, we only consider the case of $x = 0$. The case of $x > 0$ is proved similarly. Let $S = (t_0 = s_0, s_1, \ldots, s_\ell = t_1)$ be the sub-path of A, and $S' = (t_0 = s'_0, s'_1, \ldots, s'_{\ell'} = t_1)$ be the sub-path of B. Given a sequence X, we denote by $X[i, j]$ its consecutive subsequence from the i-th element to the j-th one in X.

We prove that for any $0 \leq j \leq \ell$, there exists a node $s_{c(j)} \in S$ such that $c(j) \geq j$, $\mathrm{dist}_{G_i}(t_0, s_{c(j)}) \leq kj$ and $N^+(s_{c(j)}) \cup S' \neq \emptyset$ hold. The lemma is obtained by setting $j = \ell$ because then $s_{c(j)} = s_\ell = t_1$ holds. The proof follows the induction on j. (Basis) If $j = 0$, then it holds for $s_{c(j)} = s_0$. (Inductive step) Suppose as the induction hypothesis that there exists a node $s_{c(j)}$ satisfying $c(j) \geq j$ and $\mathrm{dist}_{G_i}(t_0, s_{c(j)}) \leq kj$. If $c(j) > j$, obviously $s_{c(j+1)} = s_{c(j)}$ satisfies the case of $j + 1$. Thus, it suffices to consider the case of $c(j) = j$. Let s'_h be the neighbor of $s_{c(j)}$ in S' maximizing h, and $e = (s_{c(j)}, s'_h)$. We consider the cycle C consisting of $S[c(j), \ell]$, $S'[h, \ell']$, and e. If the length of C is at most k, obviously we have $\ell' - h \leq k - 1$. Since $\mathrm{dist}_{G_i}(t_0, s_{c(j)}) \leq kj$ holds by the induction hypothesis, $s_{c(j+1)} = s_\ell$ satisfies the condition. If the length of C is larger than k, C has a chord, which connects two nodes respectively in S and S' because both S and S' are shortest paths. Let $e' = (s_y, s'_{y'})$ be such a chord making the cycle C' consisting of e, e', $S[s_{c(j)}, s_y]$, and $S'[s_h, s'_y]$ chordless (see Figure 2). Since h is the maximum, we have $y > c(j)$ because if $y = c(j)$ the edge $e'(\neq e)$ is taken as e. Due to the property of k-chordality, the length of C' is at most k, and thus the length of path $S'[h, y'] + \{e, e'\}$ from $s_{c(j)}$ to $s_{c(x)+y}$ is at most $k - 1$, that is, $\mathrm{dist}_{G_i}(s_{c(j)}, s_y) \leq k$. By the induction hypothesis, we obtain $\mathrm{dist}_{G_i}(t_0, s_y) \leq \mathrm{dist}_{G_i}(t_0, s_{c(j)}) + \mathrm{dist}_{G_i}(s_{c(j)}, s_y) \leq k(j + 1)$. Since $s'_{y'}$ is the neighbor of s_y, we have $N^+(s_y) \cup S' \neq \emptyset$. Letting $c(j + 1) = y$, we obtain the proof for $j + 1$. The lemma holds. ◀

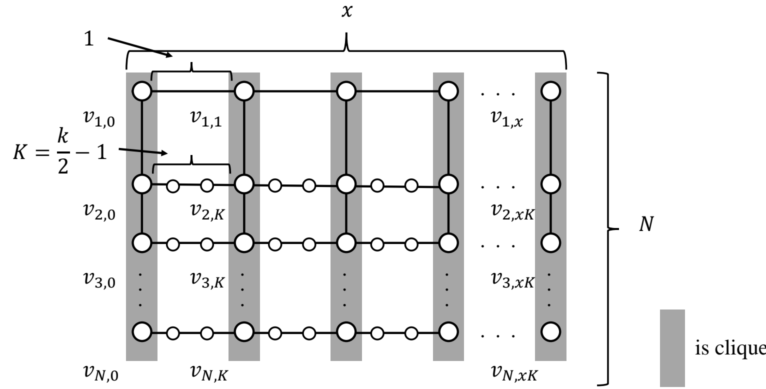


Figure 3 Example of k-chordal graph $G(k, x, N)$.

3.3 Proof of Theorem 7

We first introduce the instance mentioned in Theorem 7. Since it has two additional parameters $x \geq 0$ and $N \geq 2$ as well as k, we refer to that instance as $G(k, x, N) = (V(k, x, N), E(k, x, N))$ in the following argument. The parameters x and N are adjusted later for obtaining the claimed lower bound. Let $K = k/2 - 1$ for short. The vertex set and edge set of $G(k, x, N)$ is defined as follows:

- $V(k, x, N) = \{v_{1,j} \mid 0 \leq j \leq x\} \cup \{v_{i,j} | 2 \leq i \leq N, 0 \leq j \leq xK\}$.
- $E(k, x, N) = E_1 \cup E_2 \cup E_3 \cup E_4$ such that $E_1 = \{\{v_{1,j}, v_{1,j+1}\} \mid 0 \leq j \leq x - 1\}$, $E_2 = \{\{v_{i,j}, v_{i,j+1}\} \mid 2 \leq i \leq N, 0 \leq j \leq xK - 1\}$, $E_3 = \{\{v_{1,j}, v_{i,h}\} \mid 2 \leq i \leq N, 0 \leq j \leq x, h = jK\}$, and $E_4 = \{\{v_{i,h}, v_{j,h}\} \mid 2 \leq i, j \leq N, i \neq j, h \bmod K = 0\}$.

Figure 3 illustrates the graph $G(k, x, N)$. It is cumbersome to check this graph is k-chordal, but straightforward. One can show the following lemma.

▶ **Lemma 9.** *For $x \geq 0$ and $N \geq 2$, $G(k, x, N)$ is k-chordal.*

The proof of Theorem 7 follows the framework by Das Sarma et al. [30]. It suffices to show that the following lemma. Theorem 7 is obtained by combining this lemma with Theorem 5.

▶ **Lemma 10.** *For any $D > 2K$ and $N \geq 2kD$, $G(k, D - K, N) \in \mathcal{G}(n, N, (D - K)K + 3, 1)$ holds.*

4 Low-Congestion Shortcut for Small diameter Graphs

Let $\kappa_D = n^{(D-2)/(2D-2)}$ for short. Note that $\kappa_3 = n^{1/4}$ and $\kappa_4 = n^{1/3}$ hold. The main result in this section is the theorem below.

▶ **Theorem 11.** *For any graph of diameter $D \in \{3, 4\}$, there exists an algorithm of constructing low-congestion shortcuts with quality $\tilde{O}(\kappa_D)$ in $\tilde{O}(\kappa_D)$ rounds.*

4.1 Centralized Construction

In the following argument, we use term "whp. (with high probability)" to mean that the event considered occurs with probability $1 - n^{-\omega(1)}$ (or equivalently $1 - e^{-\omega(\log n)}$). For simplicity of the proof, we treat any whp. event as if it necessarily occurs (i.e. with probability one). Since the analysis below handles only a polynomially-bounded number of whp. events, the

standard union-bound argument guarantees that everything simultaneously occurs whp. That is, any consequence yielded by the analysis also occurs whp. Since the proof is constructive, we first present the algorithms for $D = 3$ and 4. They are described as a (unified) centralized algorithm, and the distributed implementation is explained later. Let N' be the number of parts whose diameter is more than $12\kappa_D \log^3 n$ (say *large* part). Assume that $P_1, P_2, \ldots, P_{N'}$ are large without loss of generality. Since each part P_i ($1 \leq i \leq N'$) contains at least κ_D nodes, $N' \leq n/\kappa_D$ holds obviously. The proposed algorithm constructs the shortcut edges H_i for each large part P_i following the procedure below:

1. Each node $v \in V_{P_i}$ adds its incident edges to H_i (i.e., compute the 1-hop extension).
2. This step adopts two different strategies according to the value of D. ($D = 3$) Each node $u \in N^+(V_{P_i})$ adds each incident edge (u, v) to H_i with probability $1/n^{1/2}$. ($D = 4$) Let $\mathcal{Y} = [1, n^{1/3}/\log n]$. We first prepare an $(n^{1/3}\log^3 n)$-wise independent hash function $h : [0, N-1] \times V \to \mathcal{Y}^3$. Each node $u \in V$ adds each incident edge (u, v) to H_i with probability $1/h(u, i)$ if $v \in N^+(V_{P_i})$.

We show that this algorithm provides a low-congestion shortcut of quality $\tilde{O}(\kappa_D)$. First, we look at the bound for congestion. Let H_i^1 be the set of the edges added to H_i in the first step, and H_i^2 be those in the second step. Since the congestion of 1-hop extension is negligibly small, it suffices to consider the congestion incurred by step 2. Intuitively, we can believe the congestion of $\tilde{O}(\kappa_D)$ from the fact that the expected congestion of each edge is $\tilde{O}(\kappa_D)$: Since the total number of large parts is at most n/κ_D, the expected congestion of each edge incurred in step 2 is $n/\kappa_D \cdot (1/n^{1/2}) = O(n^{1/4})$ for $D = 3$, and $(n/\kappa_D)\sum_{y \in \mathcal{Y}}(1/y) \cdot (1/|\mathcal{Y}|) \leq (n/\kappa_D) \cdot (\log n/|\mathcal{Y}|) = \tilde{O}(n^{1/3})$ for $D = 4$.

▶ **Lemma 12.** *The congestion of the constructed shortcut is $\tilde{O}(\kappa_D)$ whp.*

Proof. It suffices to show that the congestion of any edge $e = (u, v) \in E$ is $\tilde{O}(\kappa_D)$ whp. For simplicity of the proof, we see an undirected edge $e = (u, v)$ as two (directed) edges (u, v) and (v, u), and distinguish the events of adding (u, v) to shortcuts by u and that by v. That is, the former is recognized as adding (u, v), and the latter as adding (v, u). Obviously, the asymptotic bound holding for directed edge (u, v) also holds for the corresponding undirected edge (u, v) actually existing in G (which is at most twice of the directed bound). Since the first step of the algorithm increases the congestion of each directed edge at most by one, it suffices to show that the congestion incurred by the second step is at most $\tilde{O}(\kappa_D)$.

Let X_i be the indicator random variable for the event $(u, v) \in H_i^2$, and $X = \sum_i X_i$. The goal of the proof is to show that $X = \tilde{O}(\kappa_D)$ holds whp. The cases of $D = 3$ and $D = 4$ are proved separately. ($D = 3$) Since at most n/κ_3 large parts exist, we have $\mathbb{E}[X] \leq (n/\kappa_3) \cdot (1/n^{1/2}) = n^{1/4} = \kappa_3$. The straightforward application of Chernoff bound to X allows us to bound the congestion of e by at most $2\kappa_3$ with probability $1 - e^{-\Omega(n^{1/4})}$. ($D = 4$) Let $\mathcal{P}'$ be the subset of all large parts P_j such that $u \in N^+(P_j)$ holds. Consider an arbitrary partition of $\mathcal{P}'$ into several groups with size at least $(n^{1/3}\log^3 n)/2$ and at most $n^{1/3}\log^3 n$. Let q be the number of groups. Each group is identified by a number $\ell \in [1, q]$. We refer to the ℓ-th group as $\mathcal{P}^\ell$. Fixing ℓ, we bound the number of parts in $\mathcal{P}^\ell$ using $e = (u, v)$ as a shortcut edge. Let Y_i be the value of $h(u, i)$. For $P_i \in \mathcal{P}^\ell$, the probability that $X_i = 1$ is $\Pr[X_i = 1] = \sum_{y \in \mathcal{Y}} \Pr[Y_i = y] 1/y = \mathit{Har}(|\mathcal{Y}|)/|\mathcal{Y}|$, where $\mathit{Har}(x)$ is the harmonic number of x, i.e., $\sum_{1 \leq i \leq x} i^{-1}$. Letting $X^\ell = \sum_{j \in P^\ell} X_j$, we have

[3] Let X and Y be two finite sets. For any integer $k \geq 1$, a family of hash functions $\mathcal{H} = \{h_1, h_2, \ldots, h_p\}$, where each h_i is a function from X to Y, is called *k-wise independent* if for any distinct $x_1, x_2, \ldots, x_k \in X$ and any $y_1, y_2, \ldots y_k \in Y$, a function h sampled from $\mathcal{H}$ uniformly at random satisfies $\Pr[\bigwedge_{1 \leq i \leq k} h(x_i) = y_i] = 1/|Y|^k$.

$\mathbb{E}[X^\ell] = (|P^\ell| Har(|\mathcal{Y}|))/|\mathcal{Y}|$. Since $Har(x) \le \log x$, we have $(|P^\ell|\log n)/|\mathcal{Y}| \ge \mathbb{E}[X^\ell] \ge |P^\ell|/|\mathcal{Y}| = (\log^4 n)/2$. Since the hash function h is $(n^{1/3}\log^3 n)$-wise independent, it is easy to check that $X_1, X_2, \ldots, X_{p^\ell}$ are independent. We apply Chernoff bound to X^ℓ, and obtain $\Pr[X^\ell \le 2\mathbb{E}[X^\ell]] \ge 1 - e^{-\Omega(\mathbb{E}[X^\ell])} = 1 - e^{-\Omega(\log^4 n)}$. It implies that for any ℓ at most $2\mathbb{E}[X^\ell]$ groups use (u,v) as their shortcut edges. The total congestion of (u,v) is obtained by summing up $2\mathbb{E}[X^\ell]$ for all $\ell \in [1,q]$, which results in $\sum_\ell 2|P^\ell|\log n/|\mathcal{Y}| = 2|\mathcal{P}'|\log n/|\mathcal{Y}| = \tilde{O}(n^{1/3})$. The lemma is proved. ◀

For bounding dilation, we first introduce several preliminary notions and terminologies. Given a graph $G = (V,E)$, a subset $S \subset V$ is called an (α,β)-*ruling set* if it satisfies that (1) for any $u,v \in S$, $\mathrm{dist}_G(u,v) \ge \alpha$ holds, and (2) for any node $v \in V$, there exists $u \in S$ such that $\mathrm{dist}_G(v,u) \le \beta$ holds. It is known that there exists an $(\alpha, \alpha+1)$-ruling set for any graph G [2]. Let $\hat{P}_i = P_i + H_i^1$ for short. For the analysis of P_i's dilation, we first consider an $(\alpha,\alpha+1)$-ruling set of $\hat{P}_i$ for $\alpha = 12\kappa_D \log^3 n$, which is denoted by $S = \{s_0, s_1, \ldots, s_z\}$. Note that this ruling set is introduced only for the analysis, and the algorithm does not construct it actually. The key observation of the proof is that for any s_j $(1 \le j \le z)$ H_i contains a path of length $\tilde{O}(\kappa_D)$ from s_0 to s_j whp. It follows that any two nodes $u,v \in V_{\hat{P}_i}$ are connected by a path of length $\tilde{O}(\kappa_D)$ in $P_i + H_i$ because any node in $V_{\hat{P}_i}$ has at least one ruling-set node within distance $\alpha+1$ in $P_i + H_i^1$.

To prove the claim above, we further introduce the notion of *terminal sets*. A terminal set $T_j \subseteq V_{P_i}$ associated with $s_j \in S$ $(0 \le j \le z)$ is the subset of V_{P_i} satisfying (1) $|T_j| \ge \kappa_D \log^3 n$, (2) $\mathrm{dist}_{P_i+H_i}(s_j, x) \le 6\kappa_D \log^3 n$ for any $x \in T_j$, and (3) $N^+(x) \cap N^+(y) = \emptyset$ for any $x,y \in T_j$ (notice that $N^+(\cdot)$ is the set of neighbors in G, not in $P_i + H_i^1$). We can show that such a set always exists.

▶ **Lemma 13.** *Letting $S = \{s_0, s_1, \ldots, s_z\}$ be any $(\alpha,\alpha+1)$-ruling set of $\hat{P}_i$ for $\alpha = 14\kappa_D\log^3 n$, there always exists a terminal set $\mathcal{T} = \{T_0, T_1, \ldots, T_z\}$ associated with S.*

Proof. The proof is constructive. Let $c = 6\kappa_D \log^3 n$ for short. We take an arbitrary shortest path $Q = (s_j = u_0, u_1, u_2, \ldots, u_c)$ of length c in $P_i + H_i^1$ starting from $s_j \in S$. Since no two nodes in $N^+(V_{P_i}) \setminus V_{P_i}$ are adjacent in $P_i + H_i^1$, Q contains no two consecutive nodes which are both in $N^+(V_{P_i}) \setminus V_{P_i}$. It implies that at least half of the nodes in Q belongs to V_{P_i}. Let $q' = (u'_0, u'_1, \ldots u'_{c'})$ be the subsequence of Q consisting of the nodes in V_{P_i}. Then we define $T_j = \{u'_0, u'_3, \ldots, u'_{3\lfloor c'/3 \rfloor}\}$, which satisfies the three properties of terminal sets: It is easy to check that the first and second properties hold. In addition, one can show that $\mathrm{dist}_G(u'_x, u'_{x+a}) \ge 3$ (which is equivalent to $N^+(u'_x) \cap N^+(u'_{x+a}) = \emptyset$) holds for any $a \ge 3$ and $x \in [1, c'-a]$: Suppose for contradiction that $\mathrm{dist}_G(u'_x, u'_{x+a}) \le 2$ holds for some $a \ge 3$ and $x \in [1, c'-a]$. The distance two between u'_x and u'_{x+a} implies $N^+(u'_x) \cap N^+(u'_{x+a}) \ne \emptyset$, and thus $\mathrm{dist}_{\hat{P}_i}(u'_x, u'_{x+a}) \le 2$ holds. Then bypassing the subpath from u'_x to u'_{x+a} in Q through the distance-two path we obtain a path from s_j to u_c shorter than Q. It contradicts the fact that Q is the shortest path. ◀

The second property of terminal sets and the following lemma deduces the fact that $\mathrm{dist}_{P_i+H_i}(s_0, s_j) = \tilde{O}(\kappa_D)$ holds for any $j \in [0,z]$.

▶ **Lemma 14.** *Letting $S = \{s_0, s_1, \ldots, s_z\}$ be any $(\alpha,\alpha+1)$-ruling set of $\hat{P}_i$ for $\alpha = 14\kappa_D \log^3 n$, and $\mathcal{T} = \{T_0, T_1, \ldots, T_z\}$ be a terminal set associated with S. For any $j \in [0,z]$, there exist $u \in T_0$ and $v \in T_j$ such that $\mathrm{dist}_{P_i+H_i}(u,v) = O(1)$ holds.*

Proof. Since the distance of s_0 and s_j is at least $14\kappa_D \log^3 n$, we have $N^+(T_0) \cap N^+(T_j) = \emptyset$. The proof is divided into the cases of $D = 3$ and $D = 4$. $(D = 3)$ By the conditions of $N^+(T_0) \cap N^+(T_j) = \emptyset$ and $D = 3$, there exists a path of length exactly three from

any node $a \in T_0$ to any node $b \in T_j$. Letting $e_{a,b}$ be the second edge in that path, we define $F = \{e_{a,b} \mid a \in T_0, b \in T_j\}$. By the third property of terminal sets and the fact of $N^+(T_0) \cap N^+(T_j) = \emptyset$, for any two edges $(x_1, y_1), (x_2, y_2) \in F$, either $x_1 \neq x_2$ or $y_1 \neq y_2$ holds. That is, $e_{a_1,b_1} \neq e_{a_2,b_2}$ holds for any $a_1, a_2 \in T_0$ and $b_1, b_2 \in T_j$. By the second property of terminal sets, it implies $|F| = |T_0||T_j| \geq (\kappa_D \log^3 n)^2$. Since each edge in F is added to H_i^2 with probability $1/n^{1/2} = 1/\kappa_D^2$, the probability that no edge in F is added to H_i^2 is at most $(1 - 1/\kappa_D^2)^{(\kappa_D \log^3 n)^2} \leq e^{-\Omega(\log^6 n)}$. That is, an edge $e_{a,b}$ is added to H_i whp. and then $\mathrm{dist}_{P_i+H_i}(a, b) \leq 3$ holds. $(D = 4)$ For any node $u \in T_0$ and $v \in T_j$, there exists a path from u to v of length three or four in G. That path necessarily contains a length-two sub-path $P_2(u, v) = (a_{uv}, b_{uv}, c_{uv})$ such that $a_{uv} \in N^+(u)$ and $c_{uv} \in N^+(v)$ holds (if $P_2(u, v)$ is not uniquely determined, an arbitrary one is chosen). We call (a_{uv}, b_{uv}) and (b_{uv}, c_{uv}) the *first* and *second edges* of $P_2(u, v)$ respectively. Let $\mathcal{P}_2 = \{P_2(u, v) \mid u \in T_0, v \in T_j\}$, G' be the union of $P_2(u, v)$ for all $u \in T_0$ and $v \in T_j$, and $\mathcal{P}_2^e = \{P_2(u, v) \in \mathcal{P}_2 \mid e \in P_2(u, v)\}$ for any $e \in E_{G'}$. We first bound the size of $\mathcal{P}_2^e$. Assume that e is a first edge of some path in $\mathcal{P}_2^e$. Let $e = (a, b)$ and $u \in T_0$ be the (unique) node such that $a \in N^+(u)$ holds. Since at most $|T_j|$ paths in $\mathcal{P}_2$ can start from a node in $N^+(u)$, the number of paths in $\mathcal{P}_2$ using e as their first edges is at most $|T_j|$. Similarly, if e is the second edge of some path in $\mathcal{P}_2^e$, at most $|T_0|$ paths in $\mathcal{P}_2$ can contain e as their second edges. While some edge may be used as both first and second edges, the total number of paths using e is bounded by $|T_0|+|T_j| = 2\kappa_D \log^3 n$. It implies that any path $P_2(u, v)$ can share edges with at most $4\kappa_D \log^3 n$ edges, and thus $\mathcal{P}_2$ contains at least $|T_0||T_j|/(4\kappa_D \log^3 n + 1) \geq \kappa_D \log^3 n/5$ edge-disjoint paths. Let $\mathcal{P}_2' \subseteq \mathcal{P}_2$ be the maximum-cardinality subset of $\mathcal{P}_2$ such that any $P_2(u_1, v_1), P_2(u_2, v_2) \in \mathcal{P}_2'$ is edge-disjoint. We define $B = \{b \mid (a, b, c) \in \mathcal{P}_2'\}$. Let $\Delta(b)$ be the number of paths in $\mathcal{P}_2'$ containing $b \in B$ as the center. Due to the edge disjointness of $\mathcal{P}_2'$, we have $|E_G(N^+(T_0), b)| \geq \Delta(b)$ and $|E_G(N^+(T_j), b)| \geq \Delta(b)$ for any $b \in B$. Let Y_b be the value of $h(b, i)$, and X_b be the indicator random variable that takes one if a path in $\mathcal{P}_2'$ which contains b as the center is added to H_i, and zero otherwise. Let X and Y be the indicator random variables corresponding to the events of $\bigvee_{b\in B} X_b = 1$ and $\bigvee_{b\in B} Y_b \leq \Delta(b)/\log^2 n$ respectively. Then we obtain $\Pr[X_b = 1 \mid Y_b = y] \geq 1 - (1 - 1/y)^{\Delta(b)} \geq 1 - 2e^{-\Delta(b)/y}$, and thus $\Pr[X_b = 1 \mid Y_b \leq \Delta(b)/\log^2 n] \geq 1 - e^{-\Omega(\log^2 n)}$ holds. That is, $\Pr[X = 1 \mid Y = 1] \geq 1 - e^{-\Omega(\log^2 n)}$ holds. Since h is $(n^{1/3} \log^3 n)$-wise independent, Y_b for all $b \in B$ are independent. Thus we obtain

$$
\begin{aligned}
\Pr[Y = 1] &= 1 - \Pr[Y = 0] \\
&= 1 - \Pr\left[\bigwedge_{b\in B} Y_b > \frac{\Delta(b)}{\log^2 n}\right] \\
&= 1 - \prod_{b\in B} \Pr\left[Y_b > \frac{\Delta(b)}{\log^2 n}\right] \\
&= 1 - \prod_{b\in B} \left(1 - \frac{\Delta(b)}{n^{\frac{1}{3}} \log n}\right) \\
&\geq 1 - e^{-\sum_{b\in B} \frac{\Delta(b)}{n^{\frac{1}{3}} \log n}} \\
&= 1 - e^{-\frac{|\mathcal{P}_2'|}{n^{\frac{1}{3}} \log n}} \\
&\geq 1 - e^{-\Omega(\log^2 n)}.
\end{aligned}
$$

Consequently, we have $\Pr[X = 1] \geq \Pr[X = 1 \wedge Y = 1] \Pr[Y = 1] \geq \left(1 - e^{-\Omega(\log n)}\right)^2$. The lemma is proved. ◀

4.2 Distributed Implementation

We explain below the implementation details of the algorithm stated above in the CONGEST model.

- (**Preprocessing**) In the algorithm stated above, the shortcut construction is performed only for large parts, which is crucial to bound the congestion of each edge. Thus, as a preprocessing task, each node has to know if its own part is large (i.e. having a diameter larger than κ_D) or not. While the exact identification of the diameter is usually a hard task, just an asymptotic identification is sufficient for achieving the shortcut quality stated above, where the parts of diameter $\omega(\kappa_D)$ and diameter $o(\kappa_D)$ must be identified as large and small ones, but those of diameter $\Theta(\kappa_D)$ is identified arbitrarily. This loose identification is easily implemented by a simple distance-bounded aggregation. The algorithm for part P_i is that: (1)At the first round, each node in P_i sends its ID to all the neighbors, and (2)in the following rounds, each node forwards the minimum ID it received so far. The algorithm executes this message propagation during κ_D rounds. If the diameter is (substantially) larger than κ_D, the minimum ID in P_i does not reach all the nodes in P_i. Then there exists an edge whose endpoints identify different minimum IDs. The one-more-round propagation allows those endpoints to know the part is large. Then they start to broadcast the signal "large" using the following κ_D rounds. If κ_D is large, the signal "large" is invoked at several nodes in P_i, and κ_D-round propagation guarantees that every node receives the signal. That is, any node in P_i identifies that P_i is large. The running time of this task is $O(\kappa_D)$ rounds.
- (**Step 1**) As we stated, the 1-hop extension is implemented in one round. In this step, each node $v \in V_{P_i}$ tells all the neighbors if P_i is large or not. Consequently, if part P_i is identified as a large one, all the nodes in $N^+(P_i)$ know it after this step.
- (**Step 2**) The algorithm for $D = 3$ is trivial. For $D = 4$, there are two non-trivial matters. The first one is the preparation of hash function h. We realize it by sharing a random seed of $O(n^{1/3} \log^3 n \log |\mathcal{Y}|)$-bit length in advance. A standard construction by Wegman and Carter [31] allows each node to construct the desired h in common. Sharing the random seed is implemented by the broadcast of one $O(n^{1/3} \log^3 n \log |\mathcal{Y}|)$-bit message, i.e., taking $\tilde{O}(\kappa_D)$ rounds. The second matter is to address the fact that u does not know if P_i is large or not, and/or if v belongs to $N^+(P_i)$ or not. It makes u difficult to determine if (u, v) should be added to H_i or not. Instead, our algorithm simulates the task of u by the nodes in $N(u)$. More precisely, each node $v \in N^+(V_{P_i})$ adds each incident edge (u, v) to H_i with probability $1/h(u, i)$. Due to the fact of $v \in N^+(P_i)$, v knows if P_i is large or not (informed in step 1), and also can compute $h(u, i)$ locally. Thus the choice of (u, v) is locally decidable at v. Since this simulation is completely equivalent to the centralized version, the analysis of the quality also applies.

It is easy to check that the construction time of the distributed implementation above is $\tilde{O}(\kappa_D)$ in total.

5 Low-Congstion Shortcut for Bounded Clique-width Graphs

Let $G = (V, E)$ a graph. A k-graph ($k \geq 1$) is a graph whose vertices are labeled by integers in $[1, k]$. A k-graph is naturally defined as a triple (V, E, f), where f is the labeling function $f : V \to [1, k]$. The clique-width of $G = (V, E)$ is the minimum k such that there exists a k-graph $G = (V, E, f)$ which is constructed by means of repeated application of the following four operations: (1) introduce: create a graph of a single node v with label $i \in [1, k]$, (2)

disjoint union: take the union $G \cup H$ of two k-graphs G and H, (3) relabel: given $i, j \in [1, k]$, change all the labels i in the graph to j, and (4) join: given $i, j \in [1, k]$, connect all vertices labeled by i with all vertices labeled by j by edges.

The clique-width is invented first as a parameter to capture the tractability for an easy subclass of high treewidth graphs [3, 4]. That is, the class of bounded clique-width can contain many graphs with high treewidth. In centralized settings, one can often obtain polynomial-time algorithms for many NP-complete problems under the assumption of bounded clique-width. The following negative result, however, states that bounding clique-width does not admit any good solution for the MST problem (and thus also for the low-congestion shortcut).

▶ **Theorem 15.** *There exists an unweighted n-vertex graph $G = (V, E)$ of clique-width six where for any MST algorithm A there exists an edge-weight function $w_A : E \to \mathbb{N}$ such that the running time of A becomes $\tilde{\Omega}(\sqrt{n} + D)$ rounds.*

We introduce the instance stated in this theorem, which is denoted by $G(\Gamma, p)$ (Γ and p are the parameters fixed later), using the operations specified in the definition of clique-width. That is, this introduction itself becomes the proof of clique-width six. Let $\mathcal{G}(\Gamma)$ be the set of 6-graphs that contains one node with label 1, Γ nodes with label 2, and Γ nodes label 3, and all other nodes are labeled by 4. Then we define the binary operation $\oplus$ over $\mathcal{G}(\Gamma)$. For any $G, H \in \mathcal{G}(\Gamma)$, the graph $G \oplus H$ is defined as the one obtained by the following operations: (1) Relabel 2 in G with 5 and relabel 3 in H with 6, (2) take the disjoint union $G \cup H$, (3) joins with labels 5 and 6, (4) relabel 5 and 6 with 4, and then 1 with 5, (5) Add a node with label 1 by operation introduce (6) join with 1 and 5, and (7) relabel 5 with 4. This process is illustrated in Figure 4.

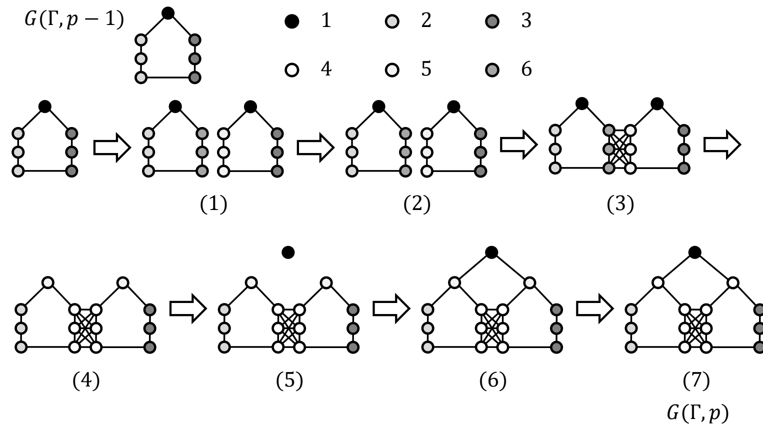


Figure 4 Graph $G \oplus H$.

Now we are ready to define $G(\Gamma, p)$. The construction is recursive. First, we define $G(\Gamma, 1)$ as follows: (1) Prepare a (2Γ)-biclique $K_{\Gamma,\Gamma}$ where one side has label 2, and the other side has label 3. Note that two labels suffice to construct $K_{\Gamma,\Gamma}$. (2) Add three nodes with label 1, 5, and 6 by operation introduce. (3) Join with label 2 and 5, and with 3 and 6. (4) Join with label 1 and 5, and with 1 and 6. (5) Relabel 5 and 6 with 4. Then, we define $G(\Gamma, p) = G(\Gamma, p-1) \oplus G(\Gamma, p-1)$. The instance claimed in Theorem 15 is $G(\sqrt{n}, \log n/2)$, which is illustrated in Figure 5. This instance is very close to the standard hard-core instance used in the prior work (e.g., [29, 30]. See Figure 1). Thus it is not difficult to see that $\tilde{\Omega}(\sqrt{n})$-round lower bound for the MST construction also applies to $G(\sqrt{n}, \log n/2)$. It suffices to show that the following lemma. Combined with Theorem 5, we obtain Theorem 15.

▶ **Lemma 16.** $G(\Gamma, p) \in \mathcal{G}(O(\Gamma(2^p + 2)), \Gamma, 2^p + 2, 3p)$.

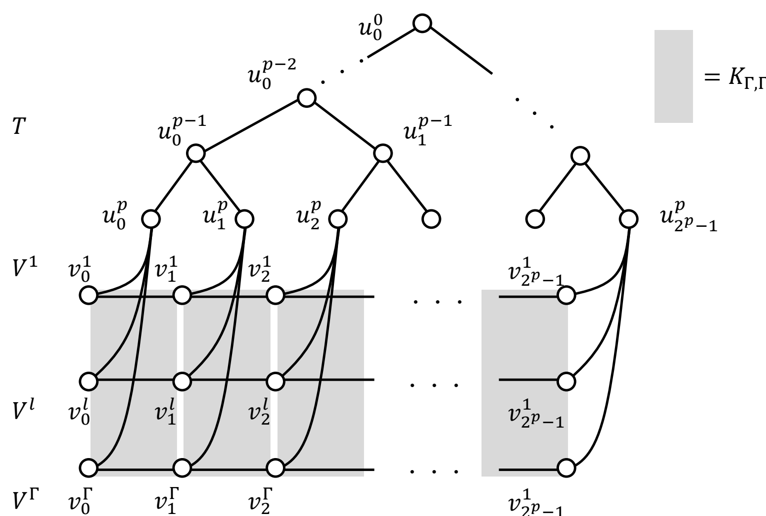


Figure 5 Example of clique-width 6 graph $G(\Gamma, p)$.

6 Conclusion

In this paper, we have shown the upper and lower bounds for the round complexity of shortcut construction and MST in k-chordal graphs, diameter-three or four graphs, and bounded clique-width graphs. We presented an $O(1)$-round algorithm constructing an optimal $O(kD)$-quality shortcut for any k-chordal graphs. We also presented the algorithms of constructing optimal low-congestion shortcuts with quality $\tilde{O}(\kappa_D)$ in $\tilde{O}(\kappa_D)$ rounds for $D = 3$ and 4, which yield the optimal algorithms for MST matching the known lower bounds by Lotker et al. [24]. On the negative side, $O(1)$-clique-width does not allow us to have good shortcuts. We conclude this paper posing three related open problems. (1) Can we have good shortcuts for $D \geq 5$? (2) Can we have good shortcuts for k-clique width where $k \leq 5$? (3) While bounded clique-width does not contribute to solving MST efficiently, it seems to provide many edge-disjoint paths (not necessarily so short). Can we find any problem that can uses the benefit of bounded clique-width?

References

1 Amir Abboud, Keren Censor-Hillel, and Seri Khoury. Near-Linear Lower Bounds for Distributed Distance Computations, Even in Sparse Networks. In *Proceedings of 30nd International Symposium on Distributed Computing (DISC)*, pages 29–42, 2016. doi:10.1007/978-3-662-53426-7_3.

2 Baruch Awerbuch, Andrew V. Goldberg, Michael Luby, and Serge A. Plotkin. Network Decomposition and Locality in Distributed Computation. In *Proceedings of 30th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 364–369, 1989. doi:10.1109/SFCS.1989.63504.

3 Derek G. Corneil and Udi Rotics. On the Relationship Between Clique-Width and Treewidth. *SIAM Journal on Computing*, pages 825–847, 2005. doi:10.1137/S0097539701385351.

4 Bruno Courcelle and Stephan Olariu. Upper bounds to the clique width of graphs. *Discrete Applied Mathematics*, pages 77–114, 2000. doi:10.1016/S0166-218X(99)00184-5.

5 Michael Elkin. Distributed approximation: a survey. *ACM SIGACT News*, pages 40–57, 2004. doi:10.1145/1054916.1054931.

6 Michael Elkin. An Unconditional Lower Bound on the Time-Approximation Trade-off for the Distributed Minimum Spanning Tree Problem. *SIAM Journal on Computing*, pages 433–456, 2006. doi:10.1137/S0097539704441058.

7 Robert G. Gallager, Pierre A. Humblet, and Philip M. Spira. A Distributed Algorithm for Minimum-Weight Spanning Trees. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, pages 66–77, 1983. doi:10.1145/357195.357200.

8 Juan A. Garay, Shay Kutten, and David Peleg. A Sublinear Time Distributed Algorithm for Minimum-Weight Spanning Trees. *SIAM Journal on Computing*, pages 302–316, 1998. doi:10.1137/S0097539794261118.

9 Fănică Gavril. The intersection graphs of subtrees in trees are exactly the chordal graphs. *Journal of Combinatorial Theory, Series B*, pages 47–56, 1974. doi:10.1016/0095-8956(74)90094-X.

10 Mohsen Ghaffari. Near-Optimal Scheduling of Distributed Algorithms. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 3–12, 2015. doi:10.1145/2767386.2767417.

11 Mohsen Ghaffari and Bernhard Haeupler. Distributed Algorithms for Planar Networks II: Low-Congestion Shortcuts, MST, and Min-Cut. In *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms (SODA)*, pages 202–219, 2016. doi:10.1137/1.9781611974331.ch16.

12 Mohsen Ghaffari and Fabian Kuhn. Distributed MST and Broadcast with Fewer Messages, and Faster Gossiping. In *Proceedings of 32nd International Symposium on Distributed Computing (DISC)*, pages 30:1–30:12, 2018. doi:10.4230/LIPIcs.DISC.2018.30.

13 Mohsen Ghaffari, Fabian Kuhn, and Hsin-Hao Su. Distributed MST and Routing in Almost Mixing Time. In *Proceedings of 31nd International Symposium on Distributed Computing (DISC)*, pages 131–140, 2017. doi:10.1145/3087801.3087827.

14 Mohsen Ghaffari and Jason Li. New Distributed Algorithms in Almost Mixing Time via Transformations from Parallel Algorithms. In *Proceedings of 32nd International Symposium on Distributed Computing (DISC)*, pages 31:1–31:16, 2018. doi:10.4230/LIPIcs.DISC.2018.31.

15 Robert Gmyr and Gopal Pandurangan. Time-Message Trade-Offs in Distributed Algorithms. In *Proceedings of 32nd International Symposium on Distributed Computing (DISC)*, pages 32:1–32:18, 2018. doi:10.4230/LIPIcs.DISC.2018.32.

16 Bernhard Haeupler, D. Ellis Hershkowitz, and David Wajc. Round- and Message-Optimal Distributed Graph Algorithms. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 119–128, 2018. doi:10.1145/3212734.3212737.

17 Bernhard Haeupler, Taisuke Izumi, and Goran Zuzic. Low-Congestion Shortcuts without Embedding. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 451–460, 2016. doi:10.1145/2933057.2933112.

18 Bernhard Haeupler, Taisuke Izumi, and Goran Zuzic. Near-Optimal Low-Congestion Shortcuts on Bounded Parameter Graphs. In *Proceedings of 30nd International Symposium on Distributed Computing (DISC)*, pages 158–172, 2016. doi:10.1007/978-3-662-53426-7_12.

19 Bernhard Haeupler and Jason Li. Faster Distributed Shortest Path Approximations via Shortcuts. In *Proceedings of 32nd International Symposium on Distributed Computing (DISC)*, pages 33:1–33:14, 2018. doi:10.4230/LIPIcs.DISC.2018.33.

20 Bernhard Haeupler, Jason Li, and Goran Zuzic. Minor Excluded Network Families Admit Fast Distributed Algorithms. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 465–474, 2018. doi:10.1145/3212734.3212776.

21 Tomasz Jurdzinski and Krzysztof Nowicki. MST in $O(1)$ rounds of congested clique. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2620–2632, 2018. doi:10.1137/1.9781611975031.167.

22 Shay Kutten and David Peleg. Fast Distributed Construction of Small k-Dominating Sets and Applications. *Journal of Algorithms*, pages 40–66, 1998. doi:10.1006/jagm.1998.0929.

23 Jason Li. Distributed Treewidth Computation. *arXiv*, 2018. arXiv:1805.10708.

24 Zvi Lotker, Boaz Patt-Shamir, and David Peleg. Distributed MST for constant diameter graphs. *Distributed Computing*, pages 453–460, 2006. doi:10.1007/s00446-005-0127-6.

25 Hiroaki Ookawa and Taisuke Izumi. Filling Logarithmic Gaps in Distributed Complexity for Global Problems. In *Proccedings of 41st International Conference on Current Trends in Theory and Practice of Informatics (SOFSEM)*, pages 377–388, 2015. doi:10.1007/978-3-662-46078-8_31.

26 Madhumangal Pal. Intersection Graphs: An Introduction. *arXiv*, 2014. arXiv:1404.5468.

27 Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. A time- and message-optimal distributed algorithm for minimum spanning trees. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 743–756, 2017. doi:10.1145/3055399.3055449.

28 Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. The Distributed Minimum Spanning Tree Problem. *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, 2018. URL: http://eatcs.org/beatcs/index.php/beatcs/article/view/538.

29 David Peleg and Vitaly Rubinovich. A Near-Tight Lower Bound on the Time Complexity of Distributed Minimum-Weight Spanning Tree Construction. *SIAM Journal on Computing*, pages 1427–1442, 2000. doi:10.1137/S0097539700369740.

30 Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. Distributed verification and hardness of distributed approximation. In *Proceedings of the 43th Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 363–372, 2011. doi:10.1145/1993636.1993686.

31 Mark N. Wegman and Larry Carter. New Hash Functions and Their Use in Authentication and Set Equality. *Journal of Computer and System Sciences*, pages 265–279, 1981. doi:10.1016/0022-0000(81)90033-7.

The Complexity of Symmetry Breaking in Massive Graphs

Christian Konrad
Department of Computer Science, University of Bristol, Bristol BS8 1UB, UK
christian.konrad@bristol.ac.uk

Sriram V. Pemmaraju
Department of Computer Science, The University of Iowa, Iowa City, IA 52242, USA
sriram-pemmaraju@uiowa.edu

Talal Riaz
Department of Computer Science, The University of Iowa, Iowa City, IA 52242, USA
sriram-pemmaraju@uiowa.edu

Peter Robinson
Department of Computer Science, City University of Hong Kong
peter.robinson@cityu.edu.hk

Abstract

The goal of this paper is to understand the complexity of symmetry breaking problems, specifically *maximal independent set (MIS)* and the closely related *β-ruling set* problem, in two computational models suited for large-scale graph processing, namely the k-machine model and the graph streaming model. We present a number of results. For MIS in the k-machine model, we improve the $\tilde{O}(m/k^2 + \Delta/k)$-round upper bound of Klauck et al. (SODA 2015) by presenting an $\tilde{O}(m/k^2)$-round algorithm. We also present an $\tilde{\Omega}(n/k^2)$ round lower bound for MIS, the first lower bound for a symmetry breaking problem in the k-machine model. For β-ruling sets, we use *hierarchical sampling* to obtain more efficient algorithms in the k-machine model and also in the graph streaming model. More specifically, we obtain a k-machine algorithm that runs in $\tilde{O}(\beta n \Delta^{1/\beta}/k^2)$ rounds and, by using a similar hierarchical sampling technique, we obtain one-pass algorithms for both insertion-only and insertion-deletion streams that use $O(\beta \cdot n^{1+1/2^{\beta-1}})$ space. The latter result establishes a clear separation between MIS, which is known to require $\Omega(n^2)$ space (Cormode et al., ICALP 2019), and β-ruling sets, even for $\beta = 2$. Finally, we present an even faster 2-ruling set algorithm in the k-machine model, one that runs in $\tilde{O}(n/k^{2-\epsilon} + k^{1-\epsilon})$ rounds for any ϵ, $0 \le \epsilon \le 1$. For a wide range of values of k this round complexity simplifies to $\tilde{O}(n/k^2)$ rounds, which we conjecture is optimal.

Our results use a variety of techniques. For our upper bounds, we prove and use simulation theorems for beeping algorithms, hierarchical sampling, and L_0-sampling, whereas for our lower bounds we use information-theoretic arguments and reductions to 2-party communication complexity problems.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases communication complexity, information theory, k-machine model, maximal independent set, ruling set, streaming algorithms

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.26

Related Version A full version of the paper is available at
https://www.dropbox.com/s/uxlwtipyydzpqsw/kprr19.pdf?dl=0.

1 Introduction

The dramatic growth in the size of graphs that need to be algorithmically processed has led to exciting research in large-scale distributed and streaming graph algorithms. Specifically, there has been a flurry of research on graph algorithms and lower bounds in models of

© Christian Konrad, Sriram V. Pemmaraju, Talal Riaz, and Peter Robinson;
licensed under Creative Commons License CC-BY

large-scale distributed computation such as the *MapReduce* model [24], the *massive parallel computation (MPC)* model [39], and the *k-machine* model [26]. Simultaneously, a lot of progress has been made on designing low-memory graph algorithms and proving memory lower bounds in different data streaming models [31]. The goal of this paper is to understand the complexity of *symmetry breaking* problems, specifically *maximal independent set (MIS)* and the closely related *ruling sets* problem, in the k-machine and streaming models. The MIS problem is a fundamental building block in distributed and parallel computing and efficient distributed algorithms for MIS are the basis for efficient distributed algorithms for problems such as *minimum dominating set* and *facility location.*[1] A *β-ruling set* of a graph $G = (V, E)$, for integer $\beta \geq 1$, is an independent set $I \subseteq V$ such that every node in V is at most β hops from some node in I. An MIS is just a 1-ruling set and β-ruling sets for larger β are natural relaxations of an MIS.

There is a rich interplay between techniques in distributed algorithms and those in streaming algorithms. On the algorithmic side, L_0-sampling [22] and the related linear graph sketches [3], which were first developed in the context of insertion-deletion streams have also been used for optimal algorithms for connectivity and MST in the distributed CongestedClique [23] and k-machine models [26, 35]. On the lower bound side, there are numerous examples of reductions to 2-party or multiparty communication complexity problems being used to derive lower bounds for both distributed computing and streaming problems. In this paper, *hierarchical sampling* is the common technical thread that connects our k-machine results and streaming results.

The k-machine model

The k-machine model was introduced by Klauck et al. [26] as an abstraction of the computation performed by large-scale graph processing systems such as Pregel [30] and Giraph (see `http://giraph.apache.org/`, [11]). This model assumes k machines $m_1, m_2, \ldots, m_k$ connected by a clique communication network. Computation and communication proceed in fault-free, synchronous rounds via message passing, as in standard models of distributed computation such as Congest [37]. The typical assumption regarding bandwidth constraints in the k-machine model is that in each round, each communication link can carry a message of size $O(\text{poly}(\log n))$ bits. The input consists of a massive n-vertex graph, with $n \gg k$. The graph is assumed to be distributed randomly in a vertex-centric fashion, i.e., each vertex and all incident edges are assigned to a machine picked uniformly at random from among the k machines. Thus each machine hosts $\tilde{O}(n/k)$ vertices *with high probability (whp)*[2]. Furthermore, a machine m_i that hosts a vertex v also knows not just the neighbors of v, but also the machines that host these neighbors. This assumption about initial knowledge is sometimes referred to as the KT1 ("(K)nowledge (T)ill Radius 1") assumption [6].

In the paper by Klauck et al. [26] and in subsequent works [7, 20, 35, 36], upper and lower bounds for several important graph problems such as *connectivity*, *minimum spanning tree (MST)*, *page rank*, *triangle enumeration*, etc., are shown. For example, an $\tilde{\Omega}(n/k^2)$ round lower bound on connectivity is shown in [26] and a tight (within logarithmic factors) upper bound of $\tilde{O}(n/k^2)$ is shown in [35]. While we have a good understanding of connectivity

---

[1] The citation for the 2016 Dijkstra prize in Distributed Computing calls MIS the "crown jewel of distributed symmetry breaking problems."

[2] We use $\tilde{O}(f(n))$ and $\tilde{\Omega}(f(n))$ notation to hide polylogarithmic factors. $\tilde{O}(f(n))$ is short for $O(f(n)\log^c n)$ for some constant c and $\tilde{\Omega}(f(n))$ is short for $\Omega(f(n)/\log^c n)$ for some constant c. The phrase "with high probability" refers to probability at least $1 - 1/n$.

and related "global" problems in the k-machine model, this understanding does not extend to MIS and symmetry breaking problems such as ruling sets. For example, [26] mention an $\tilde{O}(\min\{\frac{n}{k}, \frac{m}{k^2} + \frac{\Delta}{k}\})$-round MIS algorithm in the k-machine model that is obtained from a direct simulation of Luby's MIS algorithm [29, 4]. On the other hand, no lower bounds are known for MIS or any related symmetry breaking problems such as β-ruling sets in the k-machine model. In this paper, we shed some light on the complexity of symmetry breaking in the k-machine model; our specific results are described in more detail below.

Data Streaming Models

Data streaming algorithms are motivated by the fact that modern data sets are too large to fit into a computer's random access memory. A streaming algorithm processes its input sequentially item by item in one or few passes while maintaining a random access memory of sublinear size in the input [32]. Graph problems have been studied in the streaming model for roughly 20 years [19] (see also [31] for a more recent survey). Given an n-node graph G, a streaming algorithm processing G makes one or few passes over the edges of G. A priori no assumption is made regarding the order in which the edges "arrive". We will also consider graph streams that consist of both edge insertions and deletions (also known as *dynamic* or *turnstile* streams), where an edge can only be deleted if it has previously been inserted. This model has first been investigated by Ahn et al. [3] and has since been the focus of active research.

It is known that space $\Omega(n^2)$ space is necessary for one pass streaming algorithms [12, 5] that solve MIS, i.e., the trivial algorithm that stores all edges and computes a maximal independent set in the post-processing stage is optimal. However, nothing is known about ruling sets in the streaming model. Using a hierarchical sampling approach we show there is a β-ruling set algorithm using $o(n^2)$ space, showing a clear separation between the space complexity of β-ruling sets for $\beta = 1$ and $\beta = 2$ in the one pass streaming setting.

1.1 Main Contributions

The contributions of this paper can be organized into three categories as follows.

MIS bounds. We present an $\tilde{O}\left(\min\{\frac{n}{k}, \frac{m}{k^2}\}\right)$-round MIS algorithm in the k-machine model for graphs with m edges, improving on the $\tilde{O}\left(\min\{\frac{n}{k}, \frac{m}{k^2} + \frac{\Delta}{k}\right)$-round MIS algorithm of Klauck et al. [26]. This result follows from a more general result, namely a simulation theorem that shows that any *beeping* algorithm with message complexity msg, running in T rounds can be simulated in the k-machine model in $\tilde{O}(msg/k^2 + T)$ rounds. Beeping algorithms [1, 18, 25, 40] use extremely simple communication – just *beeps* – and node actions in a round only depend on whether a node has heard a beep (or not) in this round. Our result illustrates a general theme: algorithms in standard models of distributed computation can be automatically translated into efficient algorithms in models of large-scale distributed computing if they (i) use simple communication and (ii) if they have low round complexity *and* message complexity.

We also present an $\tilde{\Omega}(n/k^2)$ lower bound for MIS, the first non-trivial lower bound for a symmetry breaking problem in the k-machine model. Our proof starts by showing that in the 2-party communication complexity setting, there is a $O(1)$-sized graph gadget for which Alice and Bob need to communicate $\Omega(1)$ bits to find an MIS. We use an information-theoretic argument to show this and then using a direct sum type argument, we amplify this result to show an $\Omega(n)$ lower bound on the communication complexity of MIS. We then reduce the k-machine MIS problem to the 2-party MIS problem to obtain

the result, which holds even for randomized algorithms with a constant error probability. It is worth noting that this approach does not yield a 2-ruling set lower bound since 2-ruling sets can be computed in the 2-party setting without *any* communication!

Hierarchical sampling for ruling set upper bounds. We use hierarchical sampling to obtain a k-machine β-ruling set algorithm, for $\beta > 1$, that is faster than the fastest known MIS algorithm. A similar hierarchical sampling approach also leads to one-pass β-ruling set algorithms in both the insertion-only and the insertion-deletion edge-streaming models that use strictly subquadratic space. Specifically, for the β-ruling set problem, we present an $\tilde{O}(\beta \cdot n\Delta^{1/\beta}/k^2)$-round algorithm in the k-machine model and one-pass streaming algorithms using space $\tilde{O}(\beta \cdot n^{1+\frac{1}{2^\beta - 1}})$. Our k-machine β-ruling set algorithm is faster than the fastest known MIS algorithm, even for $\beta = 2$. But this result does not imply a separation between MIS and 2-ruling set in the k-machine model since we only know an $\tilde{\Omega}(n/k^2)$ lower bound for MIS. However, the streaming algorithm we present implies a clear separation between MIS and 2-ruling sets in this model due to the $\Omega(n^2)$ space lower bound for MIS [12, 5]. For insertion and deletion streams, we use L_0-*sampling* [22] as the basic building block of our algorithm.

Faster k-machine 2-ruling set algorithm. For the special case of 2-ruling sets, we present an even faster algorithm, one that runs in $\tilde{O}(n/k^{2-\epsilon} + k^{1-\epsilon})$ rounds for any ϵ, $0 \le \epsilon \le 1$. For $\epsilon = 0$, this yields an $\tilde{O}(n/k^2 + k)$-round algorithm, which simplifies to $\tilde{O}(n/k^2)$ rounds for $k \le n^{1/3}$. We conjecture that $\tilde{\Omega}(n/k^2)$ is a lower bound for 2-ruling sets in the k-machine model, and proving this would show that the above-mentioned upper bound is tight. This algorithm uses a combination of greedy-style sequential processing technique that is tailored to the k-machine model, and a beeping version of the low message complexity 2-ruling set algorithm of [34, 33] originally designed for the CONGEST model.

1.2 Related Work

The fastest MIS algorithm in the classical LOCAL and CONGEST models of distributed computin is still the three-decade old algorithm due to Luby [29] and independently due to Alon, Babai, and Itai [4]. This algorithm runs in $O(\log n)$ rounds and closing the gap between this upper bound and the $\Omega\left(\min\left\{\sqrt{\frac{\log n}{\log\log n}}, \frac{\log \Delta}{\log\log \Delta}\right\}\right)$ lower of Kuhn, Moscibroda, and Wattenhofer [28] is a major open question in this area. Assuming a bounded maximum degree, faster MIS algorithms have been very recently designed for both the LOCAL model [9, 14] and the CONGEST model [15]. β-ruling sets have also recently garnered interest in the LOCAL and CONGEST models [10, 27, 9, 10, 14, 15] and the fastest 2-ruling set algorithm in the LOCAL model breaks the Kuhn-Moscibroda-Wattenhofer lower bound and runs faster than any MIS algorithm can.

Research on algorithms and lower bounds in the k-machine model has been mentioned earlier in the introduction. The *massive parallel computation (MPC)* model is related to the k-machine model, but there are important differences in local memory and bandwidth constraints between the models. The study of classical symmetry breaking problems, especially MIS, in the MPC model is a very active area of current research [16, 17].

Many of the classic symmetry breaking problems in distributed computing have been studied in the streaming model. As mentioned earlier, there is a space $\Omega(n^2)$ lower bound for MIS for one pass algorithms [12, 5]. If multiple passes are granted, it is possible to use the correlation clustering algorithm of [2] to compute an MIS in p passes using space $\tilde{O}(n^{1+\frac{1}{2^p - 1}})$. A *maximal matching* can easily be maintained in the streaming model with space $\tilde{O}(n)$, by running the GREEDY matching algorithm. Similar to the distributed setting where computing

a $(\Delta + 1)$-coloring is easier than computing an MIS, in a recent breakthrough, Assadi et al. [5] gave a one-pass streaming algorithm with space $\tilde{O}(n)$ for $(\Delta + 1)$-coloring, even in insertion-deletion streams.

▶ Remark. Due to space constraints proofs are omitted from Sections 2.2 and 3.1. These are included in the full version of the paper.

2 Upper and Lower Bounds for MIS

2.1 An $\tilde{O}(m/k^2)$ upper bound for k-machine MIS

This section presents an $\tilde{O}(\frac{m}{k^2})$-round MIS algorithm in the k-machine model, improving on the current fastest MIS algorithm due to Klauck et al. [26] that runs in $O(\frac{m}{k^2} + \frac{\Delta}{k})$ rounds[3] The Klauck et al. MIS algorithm is simply obtained by simulating Luby's MIS algorithm in the k-machine model. Here we show a general result first, that *beeping model* algorithms [1, 18, 25, 40] can be efficiently simulated in the k-machine model and then apply this result to the $O(\log n)$-round beeping model MIS algorithm of Jeavons et al. [21].

The *beeping model* assumes a network of nodes that synchronously communicate, but only in *beeps*. A node in this model can distinguish between two situations in a round: (i) no neighbor has beeped versus (ii) at least one neighbor has beeped. The beeping model is motivated by communication in wireless networks [18, 25] and also in biological processes that solve complex problems using very simple messages e.g., neural precursor selection in the *Drosophilla* fly [1]. In both of these applications, it is found that despite the simplicity of communication, beeping model algorithms are quite powerful. Our motivation for simulating beeping algorithms in the k-machine model is similar; since a beeping algorithm has simple communication, it is easy to simulate it efficiently in the k-machine model, yet for some problems (e.g., MIS) beeping algorithms seem as powerful as algorithms that use more complex communication schemes.

To state the efficiency of our simulation, we need to define the message complexity of a beeping algorithm. Viewing each beep as a broadcast, we assume that a node v sends $degree(v)$ messages whenever it beeps. We define *message complexity*, $msg(A)$, of an algorithm A in the beeping model as the total number of messages sent during the course of the algorithm. The simulation itself is simple. Each machine performs local computations on behalf of all nodes it hosts and then sends and receives messages (beeps) on behalf of these nodes. The simulation can be done efficiently because each machine can aggregate beeps in two ways. First, if a node v hosted by machine M has several neighbors hosted by machine M', then M needs to send just one beep on v's behalf to its neighbors in M'. This aggregation works for any *broadcast* algorithm and it is exploited in the Conversion Theorem in [26]. Additional aggregation is possible because the algorithm is in the beeping model. Specifically, if M hosts several nodes $u_1, u_2, \ldots, u_p$ that have a common neighbor v hosted by M', then M can send just one beep on behalf of all of $u_1, u_2, \ldots, u_p$ to v in M'.

▶ **Theorem 1.** *A beeping algorithm A that runs in T rounds can be implemented in the k-machine model in $\tilde{O}(\frac{msg(A)}{k^2} + T)$ rounds.*

[3] Klauck et al. also point out that there is simple $\tilde{O}(n/k)$-round MIS k-machine algorithm. This allows Klauck et al. to state the running time as $\tilde{O}\left(\min\left\{\frac{n}{k}, \frac{m}{k^2} + \frac{\Delta}{k}\right\}\right)$. Our result improves this to $\tilde{O}\left(\min\left\{\frac{n}{k}, \frac{m}{k^2}\right\}\right)$.

Proof. Let a_t denote the message complexity of algorithm A in round t. A node that beeps in round t is said to be *active* in round t. (Note that a_t is the sum of the degrees of nodes that are active in round t.) Partition the active nodes in round t by their degree into $O(\log \Delta)$ degree classes; $[1,2), [2,4), [4,8), \ldots, [\Delta/2, \Delta), [\Delta, 2\Delta)$. Consider a degree class $[d, 2d)$ and let n_d denote the number of active nodes in round t in this class.

▷ Claim 2. A machine sends $\tilde{O}(\frac{a_t}{k} + k)$ messages *whp* for active nodes in degree class $[d, 2d)$ in the simulation of round t.

Proof.

Case 1: $n_d = \Omega(k \log n)$. Since each node in this class has degree at least d, the number of active nodes in this degree class is $\leq \frac{a_t}{d}$. Since nodes in the k-machine model are distributed uniformly at random among the machines, the expected number of active nodes hosted at a machine in this degree class is $\frac{a_t}{dk}$. Since $n_d = \Omega(k \log n)$, we have that $\frac{a_t}{dk} = \Omega(\log n)$. Thus, we can use a Chernoff bound to show that the number of active nodes in this degree class hosted at any machine is $O(\frac{a_t}{dk})$ *whp*. Since each node in this degree class sends at most $2d$ messages in round t, a machine needs to send at most $\tilde{O}(\frac{a_t}{dk} \cdot 2d) = \tilde{O}(\frac{a_t}{k})$ messages for this degree class *whp*.

Case 2: $n_d = O(k \log n)$. Using a Chernoff bound, we see that each machine has $O(\log n)$ active nodes from this degree class *whp*. Since A is a beeping algorithm, we need to send at most k beeps (one for each machine) for any node in the simulation of round t.

The claim follows from combining the round complexity from the two cases. ◁

Since $[d, 2d)$ is any arbitrary degree class and there are a total of $O(\log \Delta)$ degree classes, each machine sends a total of $\tilde{O}(\frac{a_t}{k} + k)$ messages to simulate round t. We can repeat the above argument by categorizing nodes by round t *in-degree*, i.e., the number of neighbors of a node that have beeped in round t. We then use the fact that in the beeping model messages incoming to a node can also be aggregated and thus a machine needs to receive at most k messages for any node it hosts. We conclude that a machine *receives* $\tilde{O}(\frac{a_t}{k} + k)$ messages *whp* in the simulation of round t.

Next, we argue that all the machines can send and receive all these messages in the k-machine model in $\tilde{O}(\frac{a_t}{k^2} + 1)$ rounds. For this we appeal to the following claim on the round complexity of a simple, randomized routing scheme (shown in Algorithm 1). In this scheme, each machine randomly selects a batch of size k messages and then distributes these to the k machines randomly as intermediate destinations. Then the intermediate nodes deterministically send these messages on to their final destinations.

▷ Claim 3. Suppose that for some positive integer X, each machine has at most X messages to send and each machine is required to receive at most X messages. Then Algorithm 1 delivers all of these messages in $\tilde{O}(X/k)$ rounds.

Proof. The number of marked messages at a machine is $\tilde{O}(k)$ *whp*. Therefore, Step 2 takes $\tilde{O}(1)$ rounds *whp*. We must now show that each machine hosts at most $\tilde{O}(1)$ messages intended for a particular destination at the beginning of Step 3.

Consider machines m_i, m_j. Again, the number of messages intended for m_i marked in Step 1 is also $\tilde{O}(k)$ *whp*. Since a message intended for m_i ends up at m_j with probability at most $1/k$, the expected number of messages intended for m_i that end up at m_j at the end of Step 2 is $\tilde{O}(1)$.

Consider a message msg that is intended for a machine m_j. Let $E_{msg,i,j}$ denote the event that message msg ends up at machine m_j at the end of Step 2. Let $X_{msg,i,j}$ denote the indicator random variable is 1 iff event $E_{msg,i,j}$ occurs, otherwise it is 0. Now, consider

another message msg'. If msg and msg' are hosted at different machines at the start of the algorithm, then the random variables $X_{msg,i,j}$ and $X_{msg',i,j}$ are independent. On the other hand, if msg and msg' were hosted at the same machine at the start of the algorithm, then the random variables $X_{msg,i,j}$ and $X_{msg',i,j}$ are negatively correlated. Then, using a Chernoff bound for negatively correlated random variables, we see that the number of messages intended for machine m_i that end up at machine m_j is $\tilde{O}(1)$ *whp* [13]. Finally, using a union bound on the total number of i, j pairs, each machine hosts $\tilde{O}(1)$ messages destined for every other machine at the end of Step 2. This means that Step 3 can be completed in $\tilde{O}(1)$ rounds, and our claim about Algorithm 1 holds. ◁

We use Algorithm 1 to route all messages in the simulation of round t $\tilde{O}(\frac{a_t}{k^2} + 1)$ rounds. Since t is any arbitrary round, we can simulate all T rounds of A in the k-machine model in $\sum_{1 \leq t \leq T} \tilde{O}(\frac{a_t}{k^2} + 1) = \tilde{O}(\frac{msg(A)}{k^2} + T)$ rounds. This completes the proof of the theorem. ◀

Algorithm 1 RandomizedRouting.

1. Each unsent message that a machine hosts is marked with probability $\min(\frac{k}{Y}, 1)$, where Y is the current number of unsent messages the machine holds.
2. Each machine distributes marked messages it holds to the k machines by picking a random permutation of these messages, and sending the ith message in the permutation to machine m_{i^*} where $i^* := i \mod k$. If a machine holds several messages intended for a particular destination, then it sends these messages one-by-one.
3. Each machine deterministically sends marked messages received in each round to their final destination. If a machine holds several messages intended for a particular destination, it will just send these messages one-by-one.

The following result is immediate by applying the simulation result above to the $O(\log n)$-round beeping model MIS algorithm of Jeavons et al. [21].

▶ **Theorem 4.** MIS *can be computed in* $\tilde{O}(\frac{m}{k^2})$ *rounds in the* k*-machine model, where* m *is the number of edges in the input graph.*

2.2 An $\tilde{\Omega}(n/k^2)$ lower bound for k-machine MIS

In this section, we show an $\tilde{\Omega}(\frac{n}{k^2})$ lower bound for MIS. While numerous lower bounds have been shown for the k-machine model for problems such as pagerank approximation, triangle enumeration, and graph connectivity (see [35, 36, 26]), these techniques cannot be applied directly to our setting. The reason for this is that the proof technique in previous work heavily relies on the fact that the input graph determines the *unique* correct solution, whereas, there are many feasible maximal independent sets for a given input graph.

In our proof we proceed as follows: We start out by considering the problem in the 2-party communication model of [38]. In particular, in Section 2.2.1 we first prove an $\Omega(1)$ communication complexity lower bound for solving MIS on a constant size gadget, which we subsequently extend to a lower bound for solving $\Theta(n)$ independent copies of the gadget. In Section 2.2.3, we describe how to extend this result to the k-machine model.

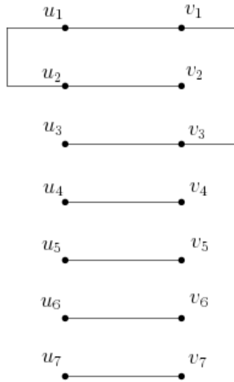


Figure 1 The lower bound gadget g with $X = 2134567$ and $Y = 3214567$.

2.2.1 A 2-party MIS Lower Bound For a Single Gadget

▶ **Theorem 5.** *The two party communication complexity of* MIS *on constant-size graphs is* $\Omega(1)$.

In the remainder of this section we prove Theorem 5. We define a 7-digit vector $s = s_1 s_2 \dots s_7$ as *valid* if each s_i is in the range $[1, 7]$ and for exactly two of its digits $s_i \neq i$. Moreover, it must be that if $s_i \neq i$ and $s_j \neq j$, then $s_i = j$ and $s_j = i$. Suppose that Alice and Bob receive inputs X and Y chosen uniformly at random from all valid 7 digit vectors. Since there are 21 such valid vectors, and X and Y are chosen uniformly at random from all valid vectors, $H[X] = H[Y] = \log_2 21$.

We will show that Alice and Bob can construct a gadget g based on the inputs X and Y such that for any MIS I of g, either the conditional mutual information, $\mathbf{I}[Y : I \mid X]$ or $\mathbf{I}[X : I \mid Y]$ is $\Omega(1)$. As it is well known that mutual information lower bounds communication complexity [8], this immediately implies the lower bound claimed in Theorem 5.

The lower bound gadget

The gadget g consists of 14 nodes, $V_A = \{u_1, \dots, u_7\}$ and $V_B = \{v_1, \dots, v_7\}$. Conceptually, the u nodes are hosted at Alice and the v nodes are hosted at Bob. The gadget contains 7 edges (u_i, v_i) for $i \in [1, 7]$. Additionally, the gadget also contains two special edges that are determined by the inputs X and Y. Based on the input $X = x_1 \dots x_7$, Alice will add a single edge between a pair of u nodes. Specifically, for the two indices i and j $(i \neq j)$ in its input vector where $x_i = j$ and $x_j = i$, Alice adds the edge (u_i, u_j). Thus, each valid vector X corresponds to a unique edge between the u nodes. Similarly, Bob will add one edge to its nodes based on its input Y. We call these two edges *special edges*. Note that except for the two special edges, the topology of the gadget is independent of the inputs X and Y.

The following lemma suffices to prove Theorem 5.

▶ **Lemma 6.** *Let* I_A *resp.* I_B *denote the vertices in the MIS output by Alice resp. Bob. Either* $\mathbf{I}[X : I_B \mid Y] = \Omega(1)$ *or* $\mathbf{I}[Y : I_A \mid X] = \Omega(1)$.

2.2.2 A 2-Party Lower Bound for Multiple Gadgets

Recalling that any pair of valid bit vectors (X, Y) uniquely defines the topology of a gadget, we call (X, Y) the *value of the gadget* and, to simplify the notation, we also use (X, Y) to refer to the gadget itself.

▶ **Theorem 7.** *The two party communication complexity of* MIS *on graphs with* $O(n)$ *nodes and edges is* $\Omega(n)$.

In the rest of this subsection, we prove Theorem 7. Consider again the two party communication complexity model where Alice receives input X and Bob receives input Y. We now consider X and Y to be vectors of length $n/2$ and we conceptually think of such a vector as the concatenation of $n/14$ 7-digit vectors as defined in Section 2.2.1. We say that an $(n/2)$-length digit vector $S = s_{1,1} \dots s_{1,7} s_{2,1} \dots s_{2,7} \dots s_{n/14,7}$ is *valid*, if $(s_{i,1} \dots s_{i,7})$ forms a *valid* 7-digit vector, for all $i \in [1, n/14]$. Let X and Y be two $n/2$ digit vectors chosen uniformly at random from all valid $(n/2)$-length digit vectors. According to the inputs X and Y, Alice and Bob will construct the lower bound graph G_L, having the property that, for any MIS I of G_L, either the mutual information between Alice's MIS output and Bob's input or between Bob's MIS output and Alice's input is $\Omega(n)$.

We now describe how to construct the lower bound graph. G_L contains n nodes partitioned into sets $V_A := \{u_1, \dots, u_{n/2}\}$ and $V_B := \{v_1, \dots, v_{n/2}\}$. All the nodes in V_A are hosted at Alice and all the nodes in V_B are hosted at Bob. The edges of G_L induce $n/14$ gadgets $\{g_1, \dots, g_{\frac{n}{14}}\}$ where gadget g_i contains nodes $u_{7i+1}, \dots, u_{7i+7}$ and $v_{7i+1}, \dots, v_{7i+7}$. The value of gadget g_i is $(x_{i,1} \dots x_{i,7}, y_{i,1} \dots y_{i,7})$. For example, gadget g_1 contains nodes $u_1, \dots, u_7$ and $v_1, \dots, v_7$, and has value $(x_{1,1} \dots x_{1,7}, y_{1,1} \dots y_{1,7})$. Notice that the topology of G_L depends only on the inputs X and Y.

Theorem 7 follows immediately from the following lemma.

▶ **Lemma 8.** *Either* $\mathbf{I}[X : I_B \mid Y] = \Omega(n)$ *or* $\mathbf{I}[Y : I_A \mid X] = \Omega(n)$.

2.2.3 Extension to the k-machine model

We are now ready to extend our results from the 2-party communication setting to the k-machine model. Specifically, we want to show a lower bound for ε error (possibly randomized) algorithms i.e., algorithms which, over all graph partitions (and random coin tosses), outputs an MIS with probability at least $(1 - \varepsilon)$, and always terminates in T rounds.

▶ **Theorem 9.** *For a constant* $\varepsilon > 0$, *any* ε*-error (possibly randomized)* MIS *algorithm in the* k*-machine model has round complexity at least* $\tilde{\Omega}(\frac{n}{k^2})$.

3 Ruling sets via hierarchical sampling

3.1 An Algorithm for the k-machine Model

In Algorithm 2 we use *hierarchical sampling* to compute a β-ruling set of an n-vertex graph with maximum degree Δ in $\tilde{O}(n\Delta^{1/\beta}/k^2)$ rounds. A hierarchical sampling approach has also been used in [10] to compute β-ruling sets and combined with the MIS algorithms of Ghaffari [14, 15], yields the fastest β-ruling set algorithms in the LOCAL and CONGEST models. But there are key differences between the LOCAL/CONGEST algorithms and our k-machine algorithm because the bandwidth constraints of the k-machine model are quite stringent (for e.g., it seems difficult for nodes that are deactivated to efficiently inform their neighbors of this fact in the k-machine model).

In each iteration i, $2 \leq i \leq \beta$, in Algorithm 2, we independently sample active nodes with probability $\Theta(\log n/\Delta^{1-(i-1)/\beta})$ (Step (3)). In each iteration, we get a sampled graph that is communicated across the k machines (Step (5)) and then we compute an MIS on this sampled graph (Step (6)). Note that in order to communicate the sampled graph, each sampled node needs to communicate with *all* neighbors and not just sampled neighbors because a priori a node does not know which neighbors have been sampled. Thus for the communication step to be efficient, i.e., complete in $\tilde{O}(n\Delta^{1/\beta}/k^2)$ rounds, as shown in Lemma 12, nodes participating the sampling step need to have relatively low degree. To ensure this high degree nodes need to be deactivated in each iteration. A node that has more than $\Delta^{1-(i-1)/\beta}$ neighbors participating in the sampling step in Iteration i is guaranteed (whp) to have a sampled neighbor and such a node will deactivate itself if it is not marked. But, a node may have high degree but with only few neighbors participating in the sampling step. A node v of this type is guaranteed to have neighbors that were deactivated in previous iterations. By inductively assuming that a node deactivated in an earlier round is not too far away from some node that has joined the ruling set, we get that node v itself is at most one extra hop away from the ruling set and can be deactivated (as in Step (7)).

Algorithm 2 RAND β-RULING SET(Graph $G = (V, E)$).

1 $P_1 \leftarrow V$
2 **for** *iteration* $i \leftarrow 2$ *to* β **do**
3 Each node in P_{i-1} marks itself with probability $\Theta(\log n/\Delta^{1-(i-1)/\beta})$.
4 $M_i \leftarrow$ nodes marked in the previous step
5 Each node in M_i informs all neighbors that it is marked
6 $I_i \leftarrow MIS(G[M_i])$
7 Each unmarked node that has a neighbor in M_i or has degree $> \Delta^{1-(i-1)/\beta}$ joins the set T_i and is *deactivated*
8 $P_i \leftarrow P_{i-1} \setminus (M_i \cup T_i)$
9 **end**
10 Each node in P_t informs neighbors that it is in P_t
11 $I \leftarrow MIS(G[P_t])$
12 **return** $(\cup_j I_j) \cup I$

▶ **Lemma 10.** *For $1 \leq i \leq \beta$, the maximum degree of nodes in P_i is at most $\Delta^{1-(i-1)/\beta}$.*

▶ **Lemma 11.** *For $2 \leq i \leq \beta$, the maximum degree of the induced graph $G[M_i]$ is at most $\tilde{O}(\Delta^{1/\beta})$ whp.*

▶ **Lemma 12.** *The communication in Steps (5) and (10) can each be completed in $\tilde{O}(n\Delta^{1/\beta}/k^2)$ rounds whp. The MIS computation in Steps (6) and (11) can each be completed in $\tilde{O}(n\Delta^{1/\beta}/k^2)$ rounds whp.*

▶ **Lemma 13.** *Every node in V is at most β hops from some node in $(\cup_{j=2}^{\beta} I_j) \cup I$.*

▶ **Theorem 14.** *For any integer $\beta \geq 1$, a β-ruling set of an n-vertex graph with maximum degree Δ can be computed in $\tilde{O}(\beta \cdot n \cdot \Delta^{1/\beta}/k^2)$ rounds.*

3.2 A One-Pass Edge-Streaming Algorithm

We now use the hierarchical sampling approach to obtain a low-memory algorithm for β-ruling sets in the edge streaming model, for $\beta > 1$. As mentioned in Section 1, our algorithm stands in contrast to the $\Omega(n^2)$ space lower bound for MIS of [12].

For each $i \in [1, \beta]$, we define

$$q_i := \frac{1}{2^{\beta - i}}. \tag{1}$$

Initially, we subsample a hierarchy of vertex sets $P_1, \ldots, P_\beta$ as follows:

- $P_1 = V(G)$.
- For $i \in [2, \beta]$, and each $u \in P_{i-1}$, we add u to P_i with probability $1/n^{q_{i-1}}$.

We call $\ell(u) = \max\{i \mid u \in P_i\}$ the *level of* u, and define the *active degree of* u as the node degree of u in the graph induced by the vertices $P_{\ell(u)} \cup \cdots \cup P_\beta$. Note that if $\ell(u) = 1$, then the active degree is simply the node degree.

Single Pass. We now describe which edges we store during the single pass of the algorithm. For each u, we store the first $\mu_{\ell(u)} := \Theta(n^{q_{\ell(u)}} \log n)$ edges that contribute to u's active degree, i.e., connect u to nodes in $P_{\ell(u)} \cup \cdots \cup P_\beta$; recall that $P_{\ell(u)} \cup \cdots \cup P_\beta \subseteq P_{\ell(u)}$. Observe that (1) implies that we store all incident edges for vertices in P_β.
Upon storing the $\mu_{\ell(u)}$-th edge for u, we mark u as *covered*. While processing the stream, we discard all edges that connect two nodes if they are both marked as covered.

Post-Processing. After the pass is completed, we move every $u \notin P_\beta$ that is *not* covered to the set P_β and set $\ell(u) = \beta$. As the last step of the algorithm, we compute an MIS S on the graph spanned by the stored edges of nodes in P_β (after post-processing) and output S as the result.

▶ **Lemma 15.** *The algorithm outputs a β-ruling set with high probability.*

Proof. We first prove that the resulting output S is indeed an independent set. To this end, we need to show that the following claim holds (after the post-processing step): For every $u, v \in P_\beta$, if there exists $(u, v) \in G$ then also $(u, v) \in E(P_\beta)$, where $E(P_\beta)$ refers to the stored edges incident to nodes in P_β: Assume towards a contradiction that the claim is false, i.e., the algorithm did not store (u, v). We distinguish 3 cases:

1. If both u and v had level β *before* post-processing, then we would have stored (u, v), as we store all incident edges for nodes in P_β.
2. Suppose only u is moved from its previous level i, whereas v was already in P_β after the initial sampling. It follows that u was not covered and hence, by definition, the edge (u, v) must have been among the first $\mu_{\ell(u)}$ edges in the stream that were incident to u and that have their other endpoint in $P_i \cup \cdots \cup P_\beta$. By the description of the algorithm, we would have stored (u, v), yielding a contradiction.
3. Finally, suppose u and v were both moved to P_β and assume (wlog) that $\ell(u) \le \ell(v)$ before the post-processing step. By a similar argument as in the previous case, it follows that we would have stored the edge (u, v) as one of the first $\mu_{\ell(u)}$ incident edges of u that point to $P_{\ell(u)} \cup \cdots \cup P_\beta$, again resulting in a contradiction.

Next, we show that the output set S satisfies the distance property of β-ruling sets, i.e., we argue that every node has distance at most β from some node in S with high probability. Recalling that we compute an MIS on the graph induced by P_β, this clearly holds for any node that has level β at this point. Thus, consider a node u with level $\ell(u) = i < \beta$ and assume that u was not moved, i.e., i continues to be the highest level of u after the post-processing step. Since $u \notin P_\beta$, we know that u was covered and hence we stored at least μ_i many edges incident to u that point to $P_i \cup \cdots \cup P_\beta$. Let $N_i(u)$ denote the corresponding set of neighbors of u for which the algorithm stores edges, i.e., $|N_i(u)| \ge \mu_i = n^{q_i} \log n$. Note that if a neighbor of u in $N_i(u)$ is moved to P_β in the post-processing step, this can only reduce

the distance of u to a node in the independent set. Therefore, it is sufficient if we show that at least one of u's neighbors in $N_i(u)$ is also part of some level greater than i. The probability that none of the μ_i nodes in $N_i(u)$ is in $P_{i+1} \cup \cdots \cup P_\beta$ is at most

$$\left(1 - \Theta\left(\frac{1}{n^{q_i}}\right)\right)^{\Theta(n^{q_i} \log n)} \leq \frac{1}{n^{\Omega(1)}}.$$

The result follows by taking a union bound over all the nodes. ◀

▶ **Lemma 16.** *The algorithm uses $O\left(\beta \cdot n^{1+1/2^{\beta-1}} \log n\right)$ space with high probability.*

Insertion-Deletion Streams

We now describe how to modify the above algorithm to work for insertion-deletion streams. The key observation is that the hierarchical sampling is done completely independently of the input stream and can be done beforehand. The only task remaining while processing the stream is storing a certain number of incident edges to a specific vertex that are also incident to a specific set. In insertion-only streams, this is straightforward. In insertion-deletion streams, we can simply use enough L_0 samplers:

Given a stream of edge insertions and deletions, an L_0-sampler is able to output a uniform random edge of the input graph (the graph obtained after all insertions and deletions have been applied). This technique can be adapted to most edge sampling tasks, such as sampling a uniform random edge incident to a specific vertex, or, by employing $\Theta(k \log n)$ L_0-samplers, sampling k different edges incident to a specific vertex, as it is required in our setting. Jowhari et al. [22] showed how to implement an L_0-sampler in insertion-deletion streams in small space:

▶ **Theorem 17** ([22])**.** *There exists an L_0 sampler for insertion-deletion streams that uses space $O(\log^2 n \log(1/\delta))$ and succeeds with probability $1 - \delta$.*

For instance, say we want to store α edges incident to a specific vertex v. Leveraging Theorem 17 tells us that we only add a polylogarithmic overhead by running $\Theta(\alpha \log n)$ L_0 samplers, and we can recover at least α different edges incident to v. Together with Lemmas 15 and 16, this implies the following result:

▶ **Theorem 18.** *In both, the insertion-only and the insertion-deletion models, there are randomized one-pass streaming algorithms with space $\tilde{O}(\beta \cdot n^{1+\frac{1}{2^\beta - 1}})$ for computing a β-ruling set that succeed with high probability.*

4 Faster 2-ruling sets in the k-machine model

The hierarchical sampling approach from Section 3.1 yields a k-machine, 2-ruling set algorithm running in $\tilde{O}(n\Delta^{1/2}/k^2)$ rounds. In this section we use a different approach to obtain a k-machine 2-ruling set algorithm that runs in $\tilde{O}(n/k^{2-\epsilon} + k^{1-\epsilon})$ rounds for any ϵ, $0 \leq \epsilon \leq 1$. Setting $\epsilon = 0$ yields an $\tilde{O}(n/k^2 + k)$-round algorithm; for $k \leq n^{1/3}$ this is an $\tilde{O}(n/k^2)$-round algorithm. The optimal value of ϵ, i.e., the value that minimizes the expression $n/k^{2-\epsilon} + k^{1-\epsilon}$, turns out to be $\epsilon = \frac{1}{2}\left(3 - \frac{\log n}{\log k}\right)$. For example, for $k = \sqrt{n}$, $\epsilon = 1/2$ is optimal and the running time simpifies to $\tilde{O}(n^{1/4})$ rounds.

Our algorithm consists of two phases. In the first phase, we perform $\lceil k^\epsilon \rceil$ iterations where we process the input graph in a sequential fashion. We say that a vertex is *active* if its MIS-status is yet undefined; otherwise we say that it is *deactivated*. In iteration $i \geq 1$ of

Phase 1, the machine m_i locally computes an MIS S_i on its part of the input and then sends S_i to all other machines using an intermediate routing step. In more detail, after computing the MIS, m_i sends the vertices in S_i in batches of size $k-1$, by transmitting the vertex IDs of the first $k-1$ vertices in S_i to the other machines over its $k-1$ links. The other machines simply relay these messages by broadcast. It is easy to see that all machines know about all nodes in S_i after m_i has sent $O(S_i/k) = \tilde{O}(n/k^2)$ batches. Before proceeding to the next iteration, each machine locally deactivates every vertex that has a neighbor in S_i.

The remaining active nodes form the residual graph and machine m_{i+1} operates on its local part of this graph in the next iteration, and so forth. After we have finished all $\lceil k^\epsilon \rceil$ iterations, each machine simply deactivates all of its vertices that are still active and have a neighbor on some machine m_j, for $j \in [1, \lceil k^\epsilon \rceil]$. In Lemma 19 below, we show that with high probability only vertices with (initial) degree $\tilde{O}(k^{1-\epsilon})$ remain active after Phase 1. We define I to be the independent set obtained by Phase 1.

For Phase 2, we use the *low message complexity* 2-RULING SET algorithm of Pai et al. [34, 33]. This algorithm runs in the CONGEST model in $O(\Delta \log n)$ rounds, with message complexity $O(n \log n)$. If we can come up with a beeping version of this 2-ruling set algorithm, then by using the Simulation Theorem (Theorem 4 in Section 2.1) we could obtain a k-machine algorithm that runs in $\tilde{O}(n/k^2 + \Delta)$ rounds. By Lemma 19, Δ is bounded above by $\tilde{O}(k^{1-\epsilon})$ after Phase 1, and thus Phase 2 would run in $\tilde{O}(\frac{n}{k^2} + k^{1-\epsilon})$ rounds. The final 2-ruling set consists of the union of set I obtained in Phase 1 and the 2-ruling set resulting from Phase 2.

Note that when starting to execute Phase 2, a machine m_j might not be aware that some of the neighbors of one of its vertices u have already been deactivated in the course of Phase 1. This does not have any effect on the round complexity of the algorithm.

This pseudocode of this two-phase algorithm is described in Algorithm 3.

Algorithm 3 TWOPHASETWORULINGSET($G = (V, E)$, ϵ).

```
   /* Phase 1: Sequential Processing                                  */
1  G_r ← G;
2  I ← ∅;
3  for i ← 1, ..., ⌈k^ε⌉ do
4  |  Machine m_i locally computes an MIS S_i on its vertices;
5  |  m_i communicates S_i to all machines;
6  |  S_i and all neighbors of nodes in S_i are removed from G_r
7  end
8  G_low ← graph induced by nodes in G_r that do not have a neighbor in any machine m_j,
     j ∈ [1, ⌈k^ε⌉];
   /* Phase 2: Low Message Complexity 2-Ruling Set                    */
9  Compute a 2-RULING SET, I' for the graph G_low using the algorithm of Pai et al. [34, 33];
10 I ← I ∪ I';
11 Return I;
```

▶ **Lemma 19.** *The time complexity of Phase 1 is $\tilde{O}(n/k^{2-\epsilon})$. Moreover, after Phase 1, the maximum vertex degree in the residual graph is $O(k^{1-\epsilon} \log n)$ with high probability.*

Proof. We first show the time complexity. Consider iteration $i \geq 1$ in which machine m_i locally computes an MIS. By the description of the algorithm, m_i sends its MIS-vertices in batches of size $O(k)$ and each machine simply broadcasts the message that it receives from m_i. Thus processing a single batch takes 2 rounds and since each batch processes $\Theta(k)$ vertices, we can complete iteration i in $\tilde{O}(n/k^2)$ rounds. The time complexity bound follows since there are $\lceil k^\epsilon \rceil$ iterations.

We next show that only vertices that have degree $O(k^{1-\epsilon} \log n)$ remain active after Phase 1. Assume in contradiction that there is some vertex u that has degree $\Omega(k^{1-\epsilon} \log n)$. Since the neighbors of u were assigned to machines using random vertex partitioning, the probability that none of them is on any of the $\lceil k^\epsilon \rceil$ machines that locally processed their vertices in Phase 1 is at most

$$\left(1 - \frac{1}{k^{1-\epsilon}}\right)^{\Omega(k^{1-\epsilon} \log n)} \le \frac{1}{n^{\Omega(1)}}.$$

Since the machine hosting u knows which machines have neighbors of u, it will deactivate u with high probability. The lemma follows by taking a union bound over all vertices. ◀

We next show that the vertices that we added when computing the local MISs in Phase 1 (and the vertices that we deactivated in the process) form a valid 2-ruling set on the induced subgraph.

▶ **Lemma 20.** *After Phase 1, the deactivated vertices form an independent set I and each deactivated vertex has distance at most 2 to some node in I.*

Proof. Clearly, no two vertices in I are neighbors since all machines deactivate neighbors of nodes that were added to the (local) MISs before proceeding to the next iteration. To see that each deactivated vertex v has distance at most 2, we distinguish two cases based on how v was deactivated. The first possibility is that v is a neighbor of some node in I and we deactivated it during some iteration, in which case the distance property trivially holds. The other possibility is that v was deactivated because it had a neighbor w on some machine m_j, for $j \in [1, \lceil k^\epsilon \rceil]$. Since m_j (locally) computed an MIS on its vertices, it follows that w has distance at most 1 to some node in I and the result follows. ◀

In Phase 2, we use the algorithm of Pai et al. [34, 33] to compute a 2-RULING SET for the graph induced by nodes that are still active and have degree at most $c \cdot \frac{n}{k^{1+\epsilon}}$ in G. In this algorithm, CATEGORY-1 refers to nodes that have joined the ruling set, CATEGORY-2 refers to their neighbors, and CATEGORY-3 refers nodes that have a CATEGORY-2, but not a CATEGORY-1 node in their neighborhood. This algorithm is similar to Luby's MIS algorithm, with two key differences to keep the message complexity low, at the cost of round complexity. First, the probability of a vertex v being marked stays *fixed* at $1/2d(v)$ and does not increase as its neighborhood shrinks. Second, a node that is marked first uses a few messsages to determine if it should deactivate itself because it has a CATEGORY-2 node in its neighborhood. This *Checking Sampling Step* plays a key role in reducing the message complexity of this algorithm to $O(n \log n)$. Below we show that this algorithm can be implemented in the k-machine model in $\tilde{O}(n/k^2 + \Delta)$ rounds.

▶ **Lemma 21.** *The message-efficient 2-ruling set algorithm of [34, 33] can be implemented in the k-machine model in $\tilde{O}(n/k^2 + \Delta)$ rounds.*

Proof. The 2-ruling set algorithm of [34, 33] is not a beeping model algorithm and we cannot apply the Simulation Theorem (Theorem 1) directly to obtain an efficient k-machine model simulation. The difficulty is caused by steps in which a node v needs to determine if a neighbor of same or higher degree has beeped. However, even in this case a machine M can aggregate all the messages going to a node v in a machine M' by simply sending only the message to v from the highest degree node it hosts. As in Theorem 1, this leads to a simulation in the k-machine model that runs in $\tilde{O}(msg/k^2 + T)$ rounds, where msg is the message complexity and T is the round complexity of the algorithm. Since Pai et al. [34, 33] have shown that msg is $O(n \log n)$ and T is $O(\Delta \log n)$, the result follows. ◀

▶ **Corollary 22.** *Phase 2 of Algorithm* TwoPhaseTwoRulingSet *can be completed in* $\tilde{O}(n/k^2 + k^{1-\epsilon})$ *rounds.*

Combining Lemmas 19, 20, and Corollary 22, we obtain an overall running time of $\tilde{O}\left(\frac{n}{k^{2-\epsilon}} + \frac{n}{k^2} + k^{1-\epsilon}\right) = O\left(\frac{n}{k^{2-\epsilon}} + k^{1-\epsilon}\right)$, which proves the following theorem:

▶ **Theorem 23.** *For any* ϵ*,* $0 \le \epsilon \le 1$*, a 2-ruling set can be computed in* $\tilde{O}\left(\frac{n}{k^{2-\epsilon}} + k^{1-\epsilon}\right)$ *rounds in the* k*-machine model.*

5 Future Work

Our results point to several natural followup questions:

1. Can we reconcile the gap between the $\tilde{O}(m/k^2)$ upper bound and $\tilde{\Omega}(n/k^2)$ lower bound for MIS? This seems related to the more fundamental question of showing tight bounds on the message complexity of MIS in the Congest KT1 model.
2. Is there an $\tilde{\Omega}(n/k^2)$ round lower bound for 2-ruling sets in the k-machine model? As pointed out earlier, our approach for the MIS lower bound that first proves an $\Omega(1)$-bit 2-party lower bound will not work for 2-ruling sets. Could an approach involving an $O(1)$-sized gadget distributed among 3 parties yield a lower bound for 2-ruling sets?
3. Can we improve the 2-ruling set upper bound to $\tilde{O}(n/k^2)$?
4. Can we prove non-trivial lower bounds on the space complexity of β-ruling sets in the one-pass edge-streaming model?

References

1. Yehuda Afek, Noga Alon, Omer Barad, Eran Hornstein, Naama Barkai, and Ziv Bar-Joseph. A Biological Solution to a Fundamental Distributed Computing Problem. *Science*, 331(6014):183–185, 2011. doi:10.1126/science.1193210.
2. Kook Jin Ahn, Graham Cormode, Sudipto Guha, Andrew McGregor, and Anthony Wirth. Correlation Clustering in Data Streams. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, pages 2237–2246, 2015. URL: http://proceedings.mlr.press/v37/ahn15.html.
3. Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Analyzing Graph Structure via Linear Measurements. In *Proceedings of the Twenty-third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, pages 459–467, Philadelphia, PA, USA, 2012. Society for Industrial and Applied Mathematics. URL: http://dl.acm.org/citation.cfm?id=2095116.2095156.
4. Noga Alon, László Babai, and Alon Itai. A Fast and Simple Randomized Parallel Algorithm for the Maximal Independent Set Problem. *J. Algorithms*, 7(4):567–583, 1986.
5. Sepehr Assadi, Yu Chen, and Sanjeev Khanna. Sublinear Algorithms for $(\Delta + 1)$ Vertex Coloring. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 767–786, 2019. doi:10.1137/1.9781611975482.48.
6. Baruch Awerbuch, Oded Goldreich, David Peleg, and Ronen Vainish. A Trade-Off between Information and Communication in Broadcast Protocols. *J. ACM*, 37(2):238–256, 1990. doi:10.1145/77600.77618.
7. Sayan Bandyapadhyay, Tanmay Inamdar, Shreyas Pai, and Sriram V. Pemmaraju. Near-Optimal Clustering in the k-machine model. In *Proceedings of the 19th International Conference on Distributed Computing and Networking, ICDCN 2018, Varanasi, India, January 4-7, 2018*, pages 15:1–15:10, 2018. doi:10.1145/3154273.3154317.

8 Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, and D. Sivakumar. An information statistics approach to data stream and communication complexity. *J. Comput. Syst. Sci.*, 68(4):702–732, 2004. doi:10.1016/j.jcss.2003.11.006.

9 Leonid Barenboim, Michael Elkin, Seth Pettie, and Johannes Schneider. The Locality of Distributed Symmetry Breaking. *J. ACM*, 63(3):20:1–20:45, June 2016. doi:10.1145/2903137.

10 Tushar Bisht, Kishore Kothapalli, and Sriram V. Pemmaraju. Brief announcement: Super-fast t-ruling sets. In *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*, pages 379–381, 2014. doi:10.1145/2611462.2611512.

11 Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One Trillion Edges: Graph Processing at Facebook-scale. *Proc. VLDB Endow.*, 8(12):1804–1815, August 2015. doi:10.14778/2824032.2824077.

12 Graham Cormode, Jacques Dark, and Christian Konrad. Independent Sets in Vertex-Arrival Streams. In *Proceedings of the 46th International Colloquium on Automata, Languages and Programming (ICALP 2019), Patras, Greece*, 2019. to appear.

13 Devdatt P. Dubhashi and Alessandro Panconesi. *Concentration of Measure for the Analysis of Randomized Algorithms*. Cambridge University Press, 2009. URL: http://www.cambridge.org/gb/knowledge/isbn/item2327542/.

14 Mohsen Ghaffari. An Improved Distributed Algorithm for Maximal Independent Set. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 270–277, 2016. doi:10.1137/1.9781611974331.ch20.

15 Mohsen Ghaffari. Distributed Maximal Independent Set using Small Messages. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 805–820, 2019. doi:10.1137/1.9781611975482.50.

16 Mohsen Ghaffari, Themis Gouleakis, Christian Konrad, Slobodan Mitrovic, and Ronitt Rubinfeld. Improved Massively Parallel Computation Algorithms for MIS, Matching, and Vertex Cover. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018, Egham, United Kingdom, July 23-27, 2018*, pages 129–138, 2018. doi:10.1145/3212734.3212743.

17 Mohsen Ghaffari and Jara Uitto. Sparsifying Distributed Algorithms with Ramifications in Massively Parallel Computation and Centralized Local Computation. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 1636–1653, 2019. doi:10.1137/1.9781611975482.99.

18 Seth Gilbert and Calvin Newport. The Computational Power of Beeps. In Yoram Moses, editor, *Distributed Computing*, pages 31–46, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

19 Monika R. Henzinger, Prabhakar Raghavan, and Sridhar Rajagopalan. Computing on Data Streams. In James M. Abello and Jeffrey Scott Vitter, editors, *External Memory Algorithms*, pages 107–118. American Mathematical Society, Boston, MA, USA, 1999. URL: http://dl.acm.org/citation.cfm?id=327766.327782.

20 Tanmay Inamdar, Shreyas Pai, and Sriram V. Pemmaraju. Large-Scale Distributed Algorithms for Facility Location with Outliers. In *22nd International Conference on Principles of Distributed Systems, OPODIS 2018, December 17-19, 2018, Hong Kong, China*, pages 5:1–5:16, 2018. doi:10.4230/LIPIcs.OPODIS.2018.5.

21 Peter Jeavons, Alex Scott, and Lei Xu. Feedback from nature: simple randomised distributed algorithms for maximal independent set selection and greedy colouring. *Distributed Computing*, 29(5):377–393, 2016. doi:10.1007/s00446-016-0269-8.

22 Hossein Jowhari, Mert Sağlam, and Gábor Tardos. Tight Bounds for Lp Samplers, Finding Duplicates in Streams, and Related Problems. In *Proceedings of the Thirtieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '11, pages 49–58, New York, NY, USA, 2011. ACM. doi:10.1145/1989284.1989289.

23 Tomasz Jurdziński and Krzysztof Nowicki. MST in O(1) Rounds of Congested Clique. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '18, pages 2620–2632, Philadelphia, PA, USA, 2018. Society for Industrial and Applied Mathematics. URL: `http://dl.acm.org/citation.cfm?id=3174304.3175472`.

24 Howard J. Karloff, Siddharth Suri, and Sergei Vassilvitskii. A Model of Computation for MapReduce. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*, pages 938–948, 2010. `doi:10.1137/1.9781611973075.76`.

25 Majid Khabbazian, Dariusz R. Kowalski, Fabian Kuhn, and Nancy Lynch. Decomposing broadcast algorithms using abstract MAC layers. *Ad Hoc Networks*, 12:219–242, 2014. `doi:10.1016/j.adhoc.2011.12.001`.

26 Hartmut Klauck, Danupon Nanongkai, Gopal Pandurangan, and Peter Robinson. Distributed Computation of Large-scale Graph Problems. In *Proceedings of the Twenty-sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '15, pages 391–410, Philadelphia, PA, USA, 2015. Society for Industrial and Applied Mathematics. URL: `http://dl.acm.org/citation.cfm?id=2722129.2722157`.

27 Kishore Kothapalli and Sriram V. Pemmaraju. Super-Fast 3-Ruling Sets. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2012, December 15-17, 2012, Hyderabad, India*, pages 136–147, 2012. `doi:10.4230/LIPIcs.FSTTCS.2012.136`.

28 Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. Local Computation: Lower and Upper Bounds. *J. ACM*, 63(2):17:1–17:44, March 2016. `doi:10.1145/2742012`.

29 M. Luby. A Simple Parallel Algorithm for the Maximal Independent Set. *SIAM Journal on Computing*, 15:1036–1053, 1986.

30 Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM. `doi:10.1145/1807167.1807184`.

31 Andrew McGregor. Graph Stream Algorithms: A Survey. *SIGMOD Rec.*, 43(1):9–20, May 2014. `doi:10.1145/2627692.2627694`.

32 S. Muthukrishnan. Data Streams: Algorithms and Applications. *Found. Trends Theor. Comput. Sci.*, 1(2):117–236, August 2005. `doi:10.1561/0400000002`.

33 Shreyas Pai, Gopal Pandurangan, Sriram V. Pemmaraju, Talal Riaz, and Peter Robinson. Brief Announcement: Symmetry Breaking in the Congest Model: Time- and Message-Efficient Algorithms for Ruling Sets. In Andréa W. Richa, editor, *31st International Symposium on Distributed Computing (DISC 2017)*, volume 91 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 38:1–38:16, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

34 Shreyas Pai, Gopal Pandurangan, Sriram V. Pemmaraju, Talal Riaz, and Peter Robinson. Symmetry Breaking in the Congest Model: Time- and Message-Efficient Algorithms for Ruling Sets. In Andréa W. Richa, editor, *31st International Symposium on Distributed Computing (DISC 2017)*, volume 91 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 38:1–38:16, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

35 Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. Fast Distributed Algorithms for Connectivity and MST in Large Graphs. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016*, pages 429–438, 2016. `doi:10.1145/2935764.2935785`.

36 Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. On the Distributed Complexity of Large-Scale Graph Computations. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures, SPAA 2018, Vienna, Austria, July 16-18, 2018*, pages 405–414, 2018. `doi:10.1145/3210377.3210409`.

37 David Peleg. *Distributed computing: a locality-sensitive approach.* Society for Industrial and Applied Mathematics, 2000.

38 Andrew Chi-Chih Yao. Some Complexity Questions Related to Distributive Computing(Preliminary Report). In *Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing*, STOC '79, pages 209–213, New York, NY, USA, 1979. ACM. doi:10.1145/800135.804414.

39 Grigory Yaroslavtsev and Adithya Vadapalli. Massively Parallel Algorithms and Hardness for Single-Linkage Clustering under ℓ_p Distances. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, pages 5596–5605, 2018. URL: http://proceedings.mlr.press/v80/yaroslavtsev18a.html.

40 J. Yu, L. Jia, D. Yu, G. Li, and X. Cheng. Minimum connected dominating set construction in wireless networks under the beeping model. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 972–980, April 2015. doi:10.1109/INFOCOM.2015.7218469.

Stellar Consensus by Instantiation

Giuliano Losa

Galois, Inc., Portland, OR, USA
http://www.losa.fr
giuliano@galois.com

Eli Gafni

UCLA, Los Angeles, CA, USA
eli@ucla.edu

David Mazières

Stanford University, CA, USA
http://www.scs.stanford.edu/~dm/addr/

Abstract

Stellar introduced a new type of quorum system called a Federated Byzantine Agreement System. A major difference between this novel type of quorum system and a threshold quorum system is that each participant has its own, personal notion of a quorum. Thus, unlike in a traditional BFT system, designed for a uniform notion of quorum, even in a time of synchrony one well-behaved participant may observe a quorum of well-behaved participants, while others may not.

To tackle this new problem in a more general setting, we abstract the Stellar Network as an instance of what we call *Personal Byzantine Quorum Systems*. Using this notion, we streamline the theory behind the Stellar Network, removing the clutter of unnecessary details, and refute the conjecture that Stellar's notion of intact set is optimally fault-tolerant. Most importantly, we develop a new consensus algorithm for the new setting.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Theory of computation → Distributed algorithms

Keywords and phrases Consensus, Stellar, Partial Synchrony, Byzantine Fault Tolerance

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.27

Related Version https://www.losa.fr/research/StellarConsensus/

Funding *Giuliano Losa*: BSF Grant 2014226, NSF Grant 1655166, a gift from the StellarDevelopment Foundation, and Galois, Inc.
Eli Gafni: BSF Grant 2014226, NSF Grant 1655166, and a gift from the Stellar Development Foundation
David Mazières: Stanford Center for Blockchain Research

Acknowledgements The authors are in debt to an anonymous reviewer who suspected that our algorithm had a flaw. Indeed, that suspicion was correct.

1 Introduction

We study the consensus problem in a new type of quorum system that we call a Personal Byzantine Quorum System (abbreviated PBQS). In a PBQS, each participant has its own, private notion of what a quorum is, subject to the requirement that if Q_p is a quorum of p and $p' \in Q_p$ then there is a quorum $Q_{p'}$ of p' inside Q_p. Justifying this rather strong requirement on the intuitive level, Q being a quorum of p has the connotation that p trusts the members of Q collectively. Hence, Q should contain at least one quorum of each $p' \in Q$.

In contrast to PBQSs, traditional Byzantine quorum systems are uniform, in the sense that a quorum is a public notion common to all participants. Under the assumptions of quorum intersection (i.e., that every two quorums intersect at a well-behaved participant)

© Giuliano Losa, Eli Gafni, and David Mazières;
licensed under Creative Commons License CC-BY

33rd International Symposium on Distributed Computing (DISC 2019).
Editor: Jukka Suomela; Article No. 27; pp. 27:1–27:15

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

and quorum availability (i.e., that at least one quorum is exclusively well-behaved), one can implement consensus under eventual synchrony [6]. However, traditionally, the ability to implement consensus using quorums is all or nothing; as soon as two quorums fail to intersect at a well-behaved participant, or if no quorum is available, no subset of the participants can solve consensus.

In a PBQS, it is possible that a subset S of the participants has intersecting quorums, in which case we say S is intertwined, while the system as a whole does not. Relying on quorum intersection to ensure safety to S is straightforward. However, suppose S_1 and S_2 are each intertwined but $S_1 \cup S_2$ is not. In this case there is no way to keep $S_1 \cup S_2$ in agreement, but we can still keep each set internally in agreement. It is also possible that $S_1 \cup S_2$ is not intertwined even though $S_1 \cap S_2 \neq \emptyset$. In this case, can a consensus algorithm also ensure liveness to S_1 and S_2? This seems impossible since, if S_1 and S_2 diverge, a participant belonging to both S_1 and S_2 has to pick a side and violate safety on the other side in order to make progress. Those observations raise the problem of determining, given an instance of PBQS and a set B of malicious participants, for which family of sets both safety and liveness are achievable, and whether there is an optimal such family. Of course, participants have no knowledge of what B is. In Section 2, we give necessary conditions for a family of sets to enjoy consensus and we define the notion of a *consensus cluster*, for which we show how to solve consensus in Section 3.

Another crucial technical difference between PBQSs and traditional Byzantine quorum systems is that since participants do not know what constitutes a quorum for another participant, even in a synchronous period, we face the asynchronous phenomenon that one well-behaved member observes a quorum of well-behaved participants, while others do not. This phenomenon was previously encountered only during periods of asynchrony.

Why is it important to study PBQSs? Beyond theoretical curiosity, PBQSs successfully abstract a deployed, real-world system: the Stellar Network. We found designing a BFT consensus algorithm which is both safe and live under these condition to be challenging. Indeed, the Stellar Consensus Protocol [14] (SCP) has only been proved non-blocking when there are Byzantine failures. Here, we propose an algorithm which is safe and live, albeit impractical. Nevertheless, it serves our purpose of showing that while the Stellar network is optimally fault-tolerant for safety, Stellar's family of intact sets, which enjoy both safety and liveness, is not optimal as previously conjectured. Furthermore, our algorithm guarantees termination in the eventually synchronous model. Whether a practical protocol can achieve these properties is still an open question.

In addition to introducing the PBQS model, we make the following contributions:

- We design an unauthenticated BFT consensus algorithm using idea from Dwork et al.[6] to solve consensus for the Stellar Network's consensus clusters.
- We refute the conjecture made in the Sellar Whitepaper [14] that intact sets are optimal for consensus. Indeed we suspect that our generalization of intact sets called consensus clusters are optimal.
- We show that the Stellar Network may harbor several disjoint consensus clusters which can nevertheless remain internally in agreement and live. Past work on federated Byzantine agreement systems [14, 7] (FBAS) assumes global quorum intersection and leaves the reader pondering whether all guarantees collapse should this assumption be violated.

Finally, we formalize the static properties of PBQSs and Stellar's federated Byzantine agreement systems in Isabelle/HOL; the formal theory is available in the Archive of Formal Proofs [12].

2 Personal Byzantine Quorum Systems

In this section we formalize the Personal Byzantine Quorum System Model (the PBQS Model), we define what it means to solve consensus in this model, we observe that global consensus is impossible even without faults, we give lower bounds on what subsets of participants can possibly enjoy consensus, and we define the notion of a consensus cluster. In a consensus algorithm, different consensus clusters may diverge, but, as we show in the next section, consensus is solvable under eventual synchrony within a consensus cluster. The main technical result of this section is that maximal consensus clusters are disjoint, as it is an obvious requirement for consensus.

▶ **Definition 1.** *A PBQS consists of a set of participants P, a set $B \subseteq P$ of Byzantine participants, a set $W = P \setminus B$ of well-behaved participants, and a function mapping a participant p to its non-empty set of quorums, which are subsets of P. The participants' quorums must be such that:*

▶ **Property 1** (Quorum sharing)**.** *If Q_p is a quorum of p and $p' \in Q_p$ then there exists a quorum $Q_{p'}$ of p' such that $Q_{p'} \subseteq Q_p$.*

In other words, Property 1 states that a quorum Q of some participant p must contain a quorum of every one of its members. As we show in Lemma 12, this remarkably simple property is sufficient to give a mathematically pleasing structure, obviously required if each consensus cluster is to be internally consistent, to PBQSs: Maximal consensus clusters are disjoint.

2.1 Consensus Algorithms in PBQSs

We assume that the participants communicate via a fully-connected point-to-point message-passing network. (In the Stellar network this is accomplished using an overlay network and signatures.) This means that a participant always knows the identity of the well-behaved sender of a message that it receives. However, message content is not authenticated (in keeping with the current Stellar Modus Operandi of not forwarding signatures) and therefore a participant cannot trust what a sender p says it heard from sender q. Well-behaved participants take steps according to the algorithm they are given, while Byzantine participants may take arbitrary steps. Each well-behaved participant is scheduled infinitely often and a message sent from a well-behaved participant to a well-behaved participant is eventually delivered.

A consensus algorithm consists of a non-terminating sequential program run by each participant in the system. The program can send and receive messages as well as take local computation steps. Initially, a participant starts with a unique identifier, a set of quorums, the set of all participants (used for round-robin leader election, which is replaced by a probabilistic election algorithm in the Stellar Network), and an input value, all of which are accessible to its program. Crucially, a participant does not know a priori the quorums of other participants (it only knows its own set of quorums). In the Stellar Network, a participant learns one of its own quorums only when it receives messages from all members of that quorum, but this difference is not of consequence. A participant also does not know which participants are Byzantine. At any point, a participant's program may produce a unique, irrevocable decision value.

▶ **Definition 2** (Intertwined)**.** *We say that a set S of well-behaved participants is intertwined when for every two sets Q and Q' which are both quorums of some (possibly different) members of S, we have $Q \cap Q' \cap W \neq \emptyset$.*

Note that, by definition, two intertwined participants cannot have empty quorums.

▶ **Definition 3** (Quorum-based algorithm). *We say that a consensus algorithm is quorum-based when:*

1. *If a well-behaved participant p decides, then there must be a quorum Q of p such that p received at least one message from each member of Q.*
2. *If Q is a quorum of a participant p, $p \in W$, and v is a possible input value, then there exists an execution in which only p and members of Q take steps, and p eventually outputs v.*

As we have already noted, a PBQS may, for example, harbor two intertwined sets S_1 and S_2 such that $S_1 \cup S_2$ is not intertwined. As implied by the following lemma, in this case no quorum-based algorithm can solve consensus for $S_1 \cup S_2$.

▶ **Lemma 4.** *Consider two participants p and p', $p \neq p'$, and two quorums Q, Q' such that Q is a quorum of p and Q' is a quorum of p' and $(Q \cap Q') \setminus B = \emptyset$. Then no quorum-based algorithm can guarantee agreement between p and p'.*

Proof. By definition of quorum-based algorithm, there are two executions e and e' such that (a) only p and members of Q take steps in e and p decides value v in e, and (b) only p' and members of Q' take steps in e' and p' decides value $v' \neq v$ in e'. Because Q and Q' are disjoint, the execution $e \cdot e'$ consisting of the concatenation of e and e' is also an execution. Moreover, agreement is violated in $e \cdot e'$. ◀

Lemma 4 shows that, in general, consensus in a PBQS is not solvable globally. Instead, we reformulate the consensus problem such that, given a PBQS $\mathcal{U}$ and a family of sets of participants depending on $\mathcal{U}$ (and thus on the quorum slices and on W), the traditional properties of consensus have to be guaranteed only to each set in the family.

▶ **Definition 5** (The PBQS Consensus Problem). *In the PBQS consensus problem for a PBQS $\mathcal{U}$ and a family of sets of participants $\{S_i\}$ (depending on $\mathcal{U}$), we require that for every set S_i in the family:*

- ***Agreement:*** *no two members of S_i decide different values.*
- ***Liveness:*** *every member of S_i eventually decides some value.*
- ***Non-triviality:*** *if only well-behaved participants take steps and a member of S_i decides, then it decides the input value of some well-behaved participant.*

Note that the definition above does not preclude any participant from taking steps in the algorithm; instead, the definition gives guarantees only to sets in the family.

In Section 2.3, we define the family of consensus clusters, and we show in Section 3 that PBQS consensus is solvable for consensus clusters. Another, more restrictive, family for which PBQS consensus is solvable is the family of intact sets, as defined in the Stellar Whitepaper. In Section 5, we show that every intact set is a consensus cluster but that the reverse is not true. In this sense, it shows that intact sets cannot be optimal for PBQS consensus. Definition 5 also raises the question of whether there exists an optimal family (in the sense of inclusion) for which PBQS consensus is solvable. We leave this question open, although we conjecture that the consensus clusters family is optimal.

2.2 A Necessary Condition for Liveness

Next we observe that if every quorum Q of a participant p contains a Byzantine node, then it is impossible to guarantee liveness for p because malicious participants can always remain silent. This is formalized using the notion of blocking set:

▶ **Definition 6** (Blocking). *If R is a set of participants, we say that p is blocked by R, or equivalently that R blocks p, when every quorum of p intersects R. We denote the set of participants blocked by R by $BlockedBy(R)$, and the set of sets that each blocks p, called p's blocking sets, by $Blocking(p)$.*

▶ **Lemma 7.** *If p is blocked by B then no quorum-based algorithm can ensure liveness to p.*

Proof. If all malicious participants remain silent, then there is no quorum Q such that p eventually receives a message from every member of Q. Therefore, by requirement 1, p never decides. ◀

An interesting question is whether q who is blocked by $BlockedBy(B)$ shares the same fate as p who is blocked by B. The answer is positive and a consequence of the quorum sharing property, as implied by the following lemma.

▶ **Lemma 8.** *In a personal quorum system, for every set of participants R, we have*

$$BlockedBy(BlockedBy(R)) = BlockedBy(R).$$

Proof. Suppose that $p \in BlockedBy(BlockedBy(R))$ but $p \notin BlockedBy(R)$. Hence, there is a quorum Q of p that does not intersect R. However, since $p \in BlockedBy(BlockedBy(R))$, Q must contain p' which is $BlockedBy(R)$. By the quorum sharing property, Q contains a quorum Q' of p', and by the virtue of p' being blocked by R, Q' contains a member of R. Since $Q' \subseteq Q$, we conclude that Q contains a member of R, and this is a contradiction. ◀

▶ **Corollary 9.** *If p is well-behaved and is not blocked by B, then p has a quorum consisting exclusively of well-behaved participants that are not blocked by B.*

2.3 Consensus Clusters

In this section we define consensus clusters and we show that maximal consensus clusters are disjoint. Consensus clusters can be thought of as disjoint islands which can be kept internally consistent and live by a consensus algorithm, but which may diverge from each other.

▶ **Definition 10** (Consensus cluster). *A subset $S \subseteq W$ of the well-behaved participants is a consensus cluster when:*

- *Quorum Intersection: S is intertwined.*
- *Quorum Availability: If $p \in S$ then there is a quorum Q_p of p such that $Q_p \subseteq S$.*

Note that, by quorum availability, a member of a consensus cluster must have a quorum, and, by quorum intersection, all its quorums must be non-empty.

We now show that maximal consensus clusters are disjoint.

▶ **Definition 11.** *A consensus cluster C is maximal when no strict superset of C is a consensus cluster.*

▶ **Lemma 12.** *Consider a personal quorum system. If C_1 and C_2 are two consensus clusters and $C_1 \cap C_2 \neq \emptyset$, then $C_1 \cup C_2$ is a consensus cluster.*

Proof. Consider $p \in C_1$ and $q \in C_2$. It suffices to show that p and q are intertwined (quorum availability is immediate). Consider two quorums Q_p and Q_q of p and q, and a quorum Q_m of a participant $m \in C_1 \cap C_2$ such that $Q_m \subseteq C_1$. Since m and q are intertwined by virtue of belonging to C_2, it follows that Q_q and Q_m have non-empty intersection in C_1. Let $n \in C_1$ be a member of this intersection. By the quorum sharing property, Q_q contains a quorum

Q_n of n. Since both n and p belong to C_1 they are intertwined. Consequently Q_p and Q_n intersect at a well-behaved participant. Since $Q_n \subseteq Q_q$, we get that Q_p and Q_q intersect at a well-behaved participant as required. ◀

▶ **Corollary 13.** *Maximal consensus clusters are disjoint.*

Finally, we present the two properties, Properties 2 and 3, that, as shown in the next section, are sufficient to solve PBQS consensus for any consensus cluster C.

▶ **Property 2** (quorum of member of C, blocks all members of C)**.** *If C is a consensus cluster and Q is a quorum of a member of C, then $Q \cap W$ blocks every member of C.*

Proof of Property 2. Consider $p \in C$. By the virtue of C being intertwined, all quorums of p intersect Q at a well-behaved participant. Thus $Q \cap W$ intersects all quorums of p, and we conclude that $Q \cap W$ blocks p. ◀

▶ **Property 3** (blocking set of member of C contains a member of C)**.** *If C is a consensus cluster, $p \in C$, and R blocks p, then $R \cap C \neq \emptyset$.*

Proof of Property 3. By definition of blocking sets, R intersects all quorums of p. Moreover, by the quorum-availability property of consensus clusters, p has a quorum $Q_p \subseteq C$. Thus, R intersects C. ◀

3 Solving Consensus under Eventual Synchrony in a PBQS

3.1 The Key Insight

Most eventually-synchronous BFT consensus algorithms [6, 3, 11, 5, 1, 8], whether they use authenticated messages or not, rely for liveness on the fact that if two participants p, p' receive the same messages then p observes a quorum (or blocking set) if and only if p' does. For example, this is used by PBFT's leader to convince other participants to prepare its value by attaching signed messages that prove that the value cannot contradict a past decision. In the unauthenticated BFT algorithm of Dwork et al. [6] (Algorithm 3), liveness is ensured by the fact that, during synchrony, a participant that locks a value at the highest round causes all other locks to be released because, thanks to reliable broadcast, the corresponding quorum is observed by all in a timely manner.

Unfortunately, those techniques fail in a PBQS because the notion of quorum is not shared by the participants: even if all participants receive the same messages, one may observe a quorum while the other does not.

The key observation that we make to solve this problem is the following. Consider a consensus cluster C. If, instead of just observing a quorum, a member p of C observes a quorum Q that unanimously states having observed a quorum making statement s, then all members of C that receive the same messages as p can derive that there is a unanimous quorum of some member of C making statement s. This is because, by Property 2, $Q \cap W$ blocks all members of C and, by Property 3, a blocking set contains a member of C, which can be trusted when it reports that a quorum of C unanimously makes statement s.

3.2 The Consensus Algorithm

We assume eventual synchrony, i.e., that there is a time GST after which (a) the messages between well-behaved participants are reliably delivered within a time bound Δ and (b) the relative rate of the clocks of any two well-behaved participants is bounded by a constant ρ. GST, Δ, and ρ are fixed but unknown to the participants.

The consensus algorithm is described in pseudocode in Algorithm 1. It consists of an unbounded sequence of rounds, where each participant progresses from round to round as instructed by a clock-synchronization protocol described in Section 3.3. The clock-synchronization protocol guarantees that there is a round GSR happening after GST such that for the round GSR and every round after GSR, members of a consensus cluster proceed from round to round synchronously, always receiving each other's messages.

Each four consecutive rounds form an *epoch*. Each epoch has a unique leader chosen round-robin. We refer to the individual rounds within an epoch as *phases*. Nodes broadcast their state at each phase. The algorithm uses a few key concepts:

- A participant *locks* a value v with an associated epoch e when it suspects that v might become decided at epoch e; if it later observes that the value was in fact not decided, then it unlocks it. Locks ensure that, within a consensus cluster, a value locked by a quorum can never be unlocked.
- A participant p considers a value *final* when it observes that no member of its consensus cluster, should p belong to a consensus cluster, can decide something different.
- A participant p *decides* a value when it observes that no participant that is intertwined with p may make a conflicting decision.
- Participants maintain a *candidate value* and keep track of the *progress-round* of their candidate; a participant assigns progress-round r to its candidate when it adopts it from the leader in round r or when it observes a unanimous quorum with the same candidate and progress round $r-1$ (we sometimes refer to progress phase when the epoch is clear from the context).

With those concepts in mind, the phases proceed as follows:

- In phase 1, a leader proposes a candidate value on which to try to agree. A node adopts the leader's value unless it suspects that a different value was decided. A node that adopts the leader's value updates its progress round to the current round.
- In phases 2 to 4, a participant p sends a candidate value to all and expects a quorum that agrees with that candidate. At each of those phases, if the expected quorum materializes, the participant updates the progress round of the candidate to the current round. The crucial property of the scheme is that, after GST, if the candidate of a member of consensus cluster C successfully progresses to phase $i > 1$, then all members of C will infer that v has progressed to phase $i-1$; this is because if Q is a quorum of a member of C, then $Q \cap W$ is a blocking set for all members of C, and a blocking sets contains a member of C and thus can be trusted.
- A participant unlocks its candidate if it gets a "proof" that, after its locking epoch, there was a quorum for another candidate. This is accomplished by observing a unanimous blocking set for a value that progressed to at least phase 2 in a higher epoch.
- A participant whose candidate progresses to phase 3 considers its candidate value *locked* (because it suspects that it may become final in phase 4), and a participant whose candidate progresses to phase 4 considers its candidate final. A final value is not revocable; in contrast "locking" is. At any time, if a participant observes a quorum unanimously declaring the same value v as final, then it decides v. While intertwined participants that are not part of a consensus cluster may disagree on final values (because those participants may be convinced to unlock arbitrarily by Byzantine participants), they cannot disagree on decisions because final is irrevocable; this is the purpose of the concept of final value.
- A participant that unlocks a value keeps a record that this value was previously locked and at what epoch, and it includes all those records in its messages. Other participants can then check that the unlock steps are valid, by making sure they can derive first-hand that a quorum justifies each unlock step; this prevents a well-behaved participants

outside a consensus cluster from "contaminating" the consensus cluster because of a bogus unlock step. This is essential because members of a consensus cluster might depend on a well-behaved outsider for quorum intersection.

The fact that final values are irrevocable guarantees that two intertwined participants never disagree. The crux of the algorithm's liveness is that a quorum with progress phase 2 suffices to unlock a value, while it takes a quorum with progress phase 3 to lock a value; this ensures that, after GST, the highest lock causes all other locks among a consensus cluster to become unlocked and the leader to adopt the corresponding value. A decision is then necessarily reached in the next epoch.

3.3 Clock Synchronization

We now describe a clock-synchronization algorithm adapted from the Stellar Consensus Protocol [14], which is simpler than the algorithm of Dwork et al. A participant p running the clock-synchronization protocol continuously advertises its current round $r[p]$ to all other participants, and it updates its round according to the following rules:

1. If p hears from a quorum whose members all advertise a round greater or equal to $r[p]$, then p arms a timer of duration $r[p] \cdot T_0$, where T_0 is some base timeout (e.g., 1 second).
2. If p's timer fires, p increments its current round.
3. If there is a round $r' > r[p]$ such that p hears from a blocking set whose members all advertise a round greater or equal to r', then p cancels any pending timeout and advances $r[p]$ to r'.

Now consider a consensus cluster C. By Property 2, rule 3 ensures that, after GST, any members of C that straggle in lower rounds catch up in constant time d_1 to the highest round that is advertised unanimously by the well-behaved portion of a quorum Q of C (because $Q \cap W$ is a blocking set for members of C). Since a blocking set must contain a member of C, rule 3 cannot be used by Byzantine participants to bring well-behaved participants to a round that was not already started by a member of C. Finally, rules 1 and 2 ensure that, despite Byzantine behavior, the first member of C to enter round r stays in round r for a duration proportional to r. Thus, round progression slows down linearly with time, and there eventually comes a round GSR after which rounds are long enough for all members of C to receive each other's messages. Note the timer duration in Rule 1 can be change, e.g., to obtain an exponential increase in round duration.

4 Consensus in Federated Byzantine Agreement Systems

In this section we show that, despite their seemingly unrealistic features, PBQSs are a useful model of Stellar's federated Byzantine agreement systems (FBASs). More precisely:

- We instantiate the consensus algorithm of Section 3 to FBASs, providing effective ways to implement its steps.
- Given a FBAS, we define a corresponding PBQS and we show that, under eventual synchrony, the instantiated consensus algorithm behaves similarly to its counterpart in the PBQS model.

The results of this section show that consensus clusters can be kept safe and live in a federated Byzantine agreement system that does not enjoy system-wide quorum intersection, whereas previous work on the subject made the assumption of system-wide quorum intersection. This is important because, in practice, misconfigured participants, rival factions, or compromised participants could, in violating quorum intersection, yield several disjoint consensus clusters.

Algorithm 1 Algorithm pseudocode.

```
struct NodeState {
    round round, progress                                        ▷ initially 1, 0
    value val                                                    ▷ initial input
    bool locked, final                                           ▷ initially false, false
    epoch lockEpoch
    set<messages> received                 ▷ all received messages with valid unlockHistory
    set<pair<epoch, value>> unlockHistory                        ▷ all values ever unlocked
}
define epoch(r) = ⌈r/4⌉
define leader(e) = participant (e mod N)                ▷ N is the number of participants
define phase(r) = r − 4 · phase(r) + 1
define valid(h) = ∀(e, v) ∈ h, a quorum sent messages w. epoch(progress) > e and val ≠ v.

method NodeState::BeginRound()
    broadcast(this)                    ▷ send (round, progress, val, final, lockHistory) to all nodes

method NodeState::EndRound()
    let F ← {m | m ∈ received and m.val = val and m.final}
    if final and {m.sender | m ∈ F} is a quorum then
        decide(val)
    end if
    if phase(round) = 1 then
        let leaderState ← m | m ∈ received and m.round = round
                 and m.sender = leader(epoch(round)))
        if !locked or leaderState.val = val then
            val ← leaderState.val
            progress ← round
        end if
    else                                                                       ▷ phases 2–4
        let R ← {m | m ∈ received and m.val = val and m.progress = round − 1}
        if {m.sender | m ∈ R} contains a quorum then
            progress ← round
            if phase(round) = 3 then
                locked ← true
                lockEpoch ← epoch(round)
            else if phase(round) = 4 then
                locked ← true
                final ← true                                       ▷ can no longer unlock
            end if
        end if
        if phase(round)= 4 then
            let B ← highest-round unanimous-val blocking set with phase(progress) = 2
            w ← unanimous value in B
            e ← epoch(round) in B
            if !locked or (locked and !final and e > lockEpoch and w ≠ val) then
                locked ← false
                unlockHistory.insert((epoch(round),val))
                val ← w                                 ▷ only matters if we are next leader
            end if
            progress ← round              ▷ sets phase(progress)= 4, not checked in phase 1
        end if
    end if
```

4.1 Federated Byzantine Agreement Systems

In a FBAS, each participant chooses a set of slices, which are sets of participants. A participant p considers a set Q to be a quorum when (a) p has at least one slice inside Q and (b) every member of Q has a slice that is a subset of Q. Practical aspects of FBASs are beyond the scope of this paper, and we refer the reader to Mazières [14] for such matters. What we will say is that it is intended that a participant will trust any information unanimously agreed upon by any of its slices, and thus a quorum is, intuitively, a set that trusts itself.

Slice-based quorums have the advantage that any new participant can join or leave the system without coordination (to join, all it needs to do is join the communication substrate; in practice, this is an overlay network emulating a point-to-point network using public-key cryptography). Moreover, any participant can also reconfigure its slices unilaterally, without coordination, e.g., to remove participants it deems unreliable or to add newcomers. On the flip side, without further assumptions, there is no guarantee that quorums will intersect, and the set of participants at a given time is generally unknown. For the analysis that follows, we assume that the set of participants is unknown but fixed and that the participants' slices do not change throughout an execution.

Three key aspects of federated Byzantine agreement systems prevent a straightforward analogy with PBQSs for the purpose of solving consensus:

1. Since each participants self-declares its set of slices (e.g., by broadcasting it), participants discover their quorums as they receive the slices of other participants. Byzantine participants have the opportunity to declare arbitrary slices and shape the quorums of well-behaved participants.
2. The algorithms of Section 3 require checking whether a set of participants is a blocking set. Doing this check by enumerating quorums is not practical even if all slices are known because the number of quorums of a participant may be exponential in the size of the system.
3. The set of participants is unknown, and thus round-robin leader-election is impossible.

4.2 Abstracting Federated Byzantine Agreement Systems

In a FBAS, participants discover quorums as they learn about the slices of other participants. Therefore, for a participant, the notion of quorum is not fixed; instead, it is augmented with new quorums as the participant learns about the slices of other participants. We call the quorums of a participant p at time t the observed quorums of p at time t. We now define a fixed notion of abstract quorums, which form a PBQS, and relate them to observed quorums.

▶ **Definition 14** (Abstract Quorums). *A set Q is an abstract quorum of participant p when $p \in B$ or p has a slice contained in Q, and every well-behaved member of Q has a slice contained in Q.*

Note that the definition of abstract quorum places requirements only on well-behaved nodes. Hence it is not computable by the participants, who do not know which participants are well-behaved. The following three lemmas are direct consequences of the definition of abstract quorum.

▶ **Lemma 15.** *Abstract quorums form a PBQS.*

Proof. From the definition of abstract quorum we immediately get that if Q is an abstract quorum of p and $p' \in Q$, then Q is an abstract quorum of p'. ◀

▶ **Lemma 16.** *If Q is an observed quorum of a well-behaved participant p at some time t, then Q is an abstract quorum.*

▶ **Lemma 17.** *Assume that Q is an abstract quorum of $p \in W$ consisting exclusively of well-behaved participants. Then, under eventual synchrony and assuming that participants do advertise their slices: shortly after GST, Q is an observed quorum of p.*

Proof. Since Q is exclusively well-behaved, shortly after GST, all well-behaved participants receive the slices of the members of Q and can check whether Q is a quorum of theirs. ◀

Lemma 16 shows that the set of abstract quorums is an over-approximation of the observed quorums. Because all the algorithms presented so far use the notion of quorum only positively (i.e. adding quorums can only enable more behaviors), Lemma 16 implies that abstract quorums are a safe abstraction of the Stellar Network when considering those algorithms, and substituting the notion of observed quorum for quorum in those algorithms does not compromise their safety properties. Lemma 17 shows that, after GST, a well-behaved participant has observed all its abstract quorums. Since the liveness of the consensus algorithm depends only on the behavior of its maximal consensus cluster, we conclude that the instantiation of the algorithm to the FBAS model preserves liveness.

4.3 Checking Whether a Set is Blocking

The algorithms of Section 3 depend on the ability for a participant p to compute whether a given set R is one of its blocking sets. Even if all slices were known, doing so by enumerating p's quorums is not practical because, by virtue of how quorums are defined in a FBAS, p may have a number of quorums that is exponential is the size of the system. Instead, we now show that there is a recursive algorithm to check whether a set is a blocking set (a) without enumerating quorums and (b) relying only on the knowledge of the slices of well-behaved participants. This algorithm can be run locally or as a distributed algorithm, e.g., as in Stellar's Federated Voting algorithm [14]. It relies on the notion of slice-blocking.

▶ **Definition 18** (Slice-Blocking)**.** *We say that the set of participants R slice-blocks p when R intersects each slice of p.*

▶ **Definition 19** (Inductively Blocked)**.** *If R is a set of participants, the set of participants inductively blocked by R, denoted R^*, is defined computationally as follows. Start with $R^* = \emptyset$. While a fixpoint is not reached, repeat the following step: add to R^* all the participants that are slice-blocked by $R^* \cup R$.*

A participant can compute locally whether some set R is blocking based on its knowledge of other's slices. However, if its knowledge of slices is incomplete, it might wrongly believe that R is not blocking. This can only remove behaviors in the algorithms of Section 3, because blocking set is used only positively, and thus, with Lemma 20, the substitution of inductively blocking for blocking does not impact safety.

Finally, Lemma 21 shows that, after GST, well-behaved blocking sets are reliably identified by well-behaved participants using the notion of inductively blocking. Thus, liveness is also preserved when substituting inductively blocking for blocking.

▶ **Lemma 20.** *At any time, if R inductively blocks $p \in W$ then R blocks p in the abstract quorum system.*

Proof. Assume by induction that if p' is in a slice of p and p' is inductively blocked by R, then all quorums of p' intersect R.

Now suppose by contradiction that R does not block p in the abstract system, i.e. that Q is an abstract quorum of p and $R \cap Q = \emptyset$. Since Q is an abstract quorum of p, there must be a slice s_p of p such that $s_p \subseteq Q$. Moreover, since R inductively blocks p, then s_p must have a member p' that is inductively blocked by R. By the quorum-sharing property, Q contains an abstract quorum of p'. Thus $Q \cap R \neq \emptyset$, which is a contradiction. ◀

▶ **Lemma 21.** *If $p \in W$ and $R \subseteq W$ blocks p in the abstract quorum system, then, shortly after GST, R inductively blocks p.*

Proof. First, observe that, shortly after GST, p knows all the slices of the well-behaved participants. Thus, suppose that p knows all the slices of the well-behaved participants.

Suppose that R does not inductively block p according to p. Then, by definition, there is a slice s_p of p whose members are not inductively blocked by R and such that $s_p \cap R = \emptyset$. Since the members of s_p are not inductively blocked by R, then, for every $p' \in s_p \setminus B$, we also have that there is a slice s'_p of p' whose members are not inductively blocked by R and such that $s'_p \cap R = \emptyset$ (we have to exclude B from s_p since p might not know the slices of Byzantine participants; in the worst case, none of those are observed inductively blocked). Continuing inductively in this fashion, we obtain an abstract quorum Q of p which does not intersect R, and we have only used the slices of well-behaved participants. This contradicts the fact that R blocks p in the abstract quorum system. ◀

4.4 Leader Election

As noted before, round-robin leader-election is impossible in a FBAS because the set of participants is in general unknown. In this section we show how to probabilistically elect a leader. However, we give no bound on the probability of success, except that it is non-zero. Devising an efficient leader-election mechanism, or, more generally, a conciliator[2] mechanism, is left open.

To agree on a common leader among C with non-zero probability, every participant p selects at random a participant p' from one of its slices or itself. If $p = p'$, then p elects itself as leader and broadcasts (leader, p). Otherwise, it waits to receive a broadcast of the form (leader, p'') from p', and then elects the participant p'' as leader and broadcasts (leader, p'').

We now show that, through this process, members of C agree on a common leader taken among C with non-zero probability.

▶ **Definition 22.** ***Graph** $D(S)$ If S is a set of participants, the directed graph $D(S)$ is defined as the graph whose set of vertices is S, and where there is an edge from n_1 to n_2 when $n_2 \neq n_1$ and n_2 is in a slice of n_1.*

▶ **Lemma 23.** *If C is a consensus cluster, $p \in C$, and Q is a quorum of a member of C, then Q is reachable from p in $D(C)$.*

Proof. Since $p \in C$ and C is a consensus cluster, there is a quorum Q' of p such that $Q' \subseteq C$. Now suppose that Q is not reachable from p in $D(C)$. Then, with $Q' \subseteq C$, we get that $Q' \cap Q = \emptyset$. This contradicts the assumption that C is a consensus cluster. ◀

▶ **Definition 24.** ***Elementary quorum** An elementary quorum is a quorum Q such that no strict subset of Q is a quorum.*

Note that, by definition, every quorum contains an elementary quorum.

▶ **Lemma 25.** *If n_1 and n_2 are members of an elementary quorum q consisting exclusively of well-behaved participants, then there is a path in $D(q)$ from n_1 to n_2.*

Proof. Suppose q is an elementary quorum and that $n_1, n_2 \in q$ and n_2 is not reachable from n_1 in $D(q)$. Then consider the set S of participants that are reachable from n_1 in $D(q)$. By our assumption above, n_2 does not belong to S. Thus S is a strict subset of q. Moreover, every member n of S has a slice $\mathrm{s}_n \subseteq q$. Additionally, consider that we must have that $\mathrm{s}_n \subseteq S$, as otherwise a participant outside S would be reachable from n_1. Thus every member of S has a slice in S, and therefore S is a quorum. Since S is a strict subset of q, this contradicts the fact that q is an elementary quorum. ◀

▶ **Lemma 26.** *If C is a consensus cluster, then there exists a member of C that is reachable in $D(C)$ from every other participant in C.*

Proof. Since C is a quorum, C contains an elementary quorum Q. By Lemma 23, Q is reachable from every member n of C in $D(C)$. Moreover, by Lemma 25, every member of Q is reachable in $D(Q)$ from every other member of Q. Thus, because $D(Q) \subseteq D(C)$, every member of Q is reachable in $D(C)$ from every member of C. ◀

▶ **Lemma 27.** *If C is a consensus cluster, then, with non-zero probability, every member of C elects the same leader $l \in C$.*

Proof. Note that the leader-election algorithm can be seen as randomly selection edges in $D(P)$ (where P is the set of participants). Because there is a member n of C reachable in D from all other members of C in $D(C)$ (and because well-behaved participant have a finite number of outgoing edges), then with non-zero probability the edges selected by the leader-election algorithm will form a sink tree rooted at n, who will be elected unique leader by all members of C. ◀

5 Related Work

Federated Byzantine quorum systems were first introduced in the Stellar Whitepaper by Mazières [14], who also proposes the notion of intact set and a consensus algorithm for intact sets, the Stellar Consensus Protocol (SCP). The epidemic propagation mechanism and the clock-synchronization protocol presented in the present paper are taken from the Stellar Whitepaper. Mazières also discusses more practical aspects of the Stellar Network.

One important contribution of the present paper is that Stellar's intact sets, conjectured in the Stellar Whitepaper to be optimal for consensus, are in fact not the biggest sets for which an algorithm can solve consensus. An intact set is a subset S of W such that every member of S is well-behaved and: (a) if Q and Q' are quorums of S, then $Q \cap Q' \cap S \neq \emptyset$; (b) S is a quorum. Comparing the definitions of consensus cluster and intact set, it is easy to see that any intact set is also a consensus cluster. However, as shown by the following lemma, there are some consensus clusters that are strictly bigger than any intact set.

▶ **Lemma 28.** *There are some configurations in which a set S is a consensus cluster but S is not intact and S has no intact superset.*

Proof. Consider a system of three well-behaved participants p_1, p_2, and p_3 (note that there are no malicious participants) where p_1 has a single slice $\{p_1\}$, p_2 has two slices $\{p_1, p_2\}$ and $\{p_2, p_3\}$, and p_3 has two slices $\{p_1, p_3\}$ and $\{p_2, p_3\}$. According to those slices, the quorums are $\{p_1\}$, $\{p_1, p_2, p_3\}$, $\{p_2, p_3\}$, $\{p_1, p_2\}$, and $\{p_1, p_3\}$. In this system, $C = \{p_2, p_3\}$

is a consensus cluster but is not intact, because $Q_1 = \{p_1, p_2\}$ and $Q_2 = \{p_1, p_3\}$ intersect outside C. Moreover, the only strict superset of C, $\{p_1, p_2, p_3\}$, is not intact because the quorums $\{p_1\}$ and $\{p_2, p_3\}$ do not intersect. ◀

Another novel aspect of the present paper compared to the Stellar Whitepaper is that we do not assume global quorum intersection; nevertheless, we show that consensus clusters enjoy safe and live consensus. This is important because it shows that safety and liveness guarantees do not collapse system-wide in the face of misconfigurations or attacks.

We have studied federated quorum system under the assumption that well-behaved participants do not change their slices. However, in practice, well-behaved participants might change their slices to eliminate unreliable participants or add newcomers. The Stellar Whitepaper also analyzes this situation.

García-Pérez and Gotsman [7] study in details Stellar's federated Byzantine quorum systems and the implementation of broadcast abstractions therein. They also propose the notion of subjective dissemination quorum system (DQS) in which, like in a PBQS, each participant has its own set of quorums. However, subjective DQSs have two crucial differences compared to PBQSs: subjective DQSs have system-wide quorum intersection and they do not have Property 1 (which says that a quorum is a quorum for all its members). In the absence of system-wide quorum intersection, Property 1 of PBQSs ensures that maximal consensus clusters are disjoint (Lemma 12). Without it, maximal consensus clusters may intersect, which implies that consensus is not solvable even for consensus clusters (a participant in the intersection may have to violate safety on one side in order to make progress).

Ripple [15] introduced the first permissionless quorum-based consensus protocol. In the XRP Ledger Consensus Protocol, each participant p is responsible for configuring its own UNL, which is a list of participants that p accepts messages from. Moreover, p considers as a quorum any set of participants consisting of more than a fixed fraction (defined system-wide by the protocol, e.g. 80%) of its UNL. Maintaining agreement in Ripple's protocol rests on the assumption that participants will provide sufficiently overlapping UNLs (roughly 90% for every pair of participants, in the most adversarial model of Chase and MacBrough [4]).

Traditional Byzantine quorum systems are uniform, in the sense that every participant has the same notion of quorum. Uniform Byzantine quorum systems are studied in details by Malkhi and Reiter [13]. More complex types of uniform quorum systems are studied by Guerraoui and Vukolić [9]. General Byzantine adversaries [10] do not give rise to a PBQS because participants have global knowledge of the adversary in this model.

References

1. Ittai Abraham, Guy Gueta, Dahlia Malkhi, and Jean-Philippe Martin. Revisiting fast practical byzantine fault tolerance: Thelma, Velma, and Zelma. *arXiv preprint*, 2018. `arXiv:1801.10022`.
2. James Aspnes. A modular approach to shared-memory consensus, with applications to the probabilistic-write model. *Distributed Computing*, 25(2):179–188, 2012.
3. Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.
4. Brad Chase and Ethan MacBrough. Analysis of the XRP Ledger consensus protocol. *arXiv preprint*, 2018. `arXiv:1802.07242`.
5. Allen Clement, Edmund L Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. In *NSDI*, volume 9, pages 153–168, 2009.

6 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

7 Álvaro García-Pérez and Alexey Gotsman. Federated Byzantine Quorum Systems. In *22nd International Conference on Principles of Distributed Systems (OPODIS 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

8 Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 BFT protocols. In *Proceedings of the 5th European conference on Computer systems*, pages 363–376. ACM, 2010.

9 Rachid Guerraoui and Marko Vukolić. Refined quorum systems. *Distributed Computing*, 23(1):1–42, 2010.

10 Martin Hirt and Ueli Maurer. Complete characterization of adversaries tolerable in secure multi-party computation. In *PODC*, volume 97, pages 25–34, 1997.

11 Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 45–58, New York, NY, USA, 2007. ACM.

12 Giuliano Losa. Stellar Quorum Systems. *Archive of Formal Proofs*, August 2019. , Formal proof development. URL: `http://isa-afp.org/entries/Stellar_Quorums.html`.

13 Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed computing*, 11(4):203–213, 1998.

14 David Mazieres. The Stellar Consensus Protocol: A federated model for internet-level consensus. *Stellar Development Foundation*, page 32, 2015.

15 David Schwartz, Noah Youngs, and Arthur Britto. The Ripple Protocol Consensus Algorithm, 2014. URL: `https://ripple.com/files/ripple_consensus_whitepaper.pdf`.

A Scalable, Portable, and Memory-Efficient Lock-Free FIFO Queue

Ruslan Nikolaev
Virginia Tech, Blacksburg, VA, USA
rnikola@vt.edu

Abstract

We present a new lock-free multiple-producer and multiple-consumer (MPMC) FIFO queue design which is scalable and, unlike existing high-performant queues, very memory efficient. Moreover, the design is ABA safe and does not require any external memory allocators or safe memory reclamation techniques, typically needed by other scalable designs. In fact, this queue itself can be leveraged for object allocation and reclamation, as in data pools. We use FAA (fetch-and-add), a specialized and more scalable than CAS (compare-and-set) instruction, on the most contended hot spots of the algorithm. However, unlike prior attempts with FAA, our queue is both lock-free and linearizable.

We propose a general approach, SCQ, for bounded queues. This approach can easily be extended to support unbounded FIFO queues which can store an arbitrary number of elements. SCQ is portable across virtually all existing architectures and flexible enough for a wide variety of uses. We measure the performance of our algorithm on the x86-64 and PowerPC architectures. Our evaluation validates that our queue has exceptional memory efficiency compared to other algorithms and its performance is often comparable to, or exceeding that of state-of-the-art scalable algorithms.

2012 ACM Subject Classification Theory of computation → Concurrent algorithms

Keywords and phrases FIFO, queue, ring buffer, lock-free, non-blocking

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.28

Related Version A full version of the paper is available at `https://arxiv.org/abs/1908.04511`.

Supplement Material SCQ's source code is available at `https://github.com/rusnikola/lfqueue`.

Acknowledgements We thank the anonymous reviewers for their valuable feedback.

1 Introduction

Creating efficient concurrent algorithms for modern multi-core architectures is challenging. Efficient and scalable lock-free FIFO queues proved to be especially hard to devise. Although elimination techniques address LIFO stack performance [6], their FIFO counterparts [18] are somewhat more restricted: a *dequeue* operation can eliminate an *enqueue* operation only if all preceding queue entries have already been consumed. Thus, FIFO elimination is more suitable in specialized cases and, typically, shorter queues. Relaxed versions of FIFO queues, which can reorder elements, were also proposed [9], but they cannot be used when a strict FIFO ordering is required. FIFO queues are important in many applications, especially in fixed-size data pools which use bounded *circular queues* (*ring buffers*).

Most correct linearizable and lock-free implementations of such queues rely on the compare-and-set (CAS) instruction. Although this instruction is powerful, it does not scale well as the contention grows. However, less powerful instructions such as fetch-and-add (FAA) are known to scale better. In Figure 1, we demonstrate execution time for FAA vs. CAS measured in a tight loop for the corresponding number of threads. The results are for an almost "ideal case" to simply emulate FAA in a CAS loop. As CAS loops in typical algorithms are more complex, the actual gap is bigger.

© Ruslan Nikolaev;
licensed under Creative Commons License CC-BY
33rd International Symposium on Distributed Computing (DISC 2019).
Editor: Jukka Suomela; Article No. 28; pp. 28:1–28:16

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

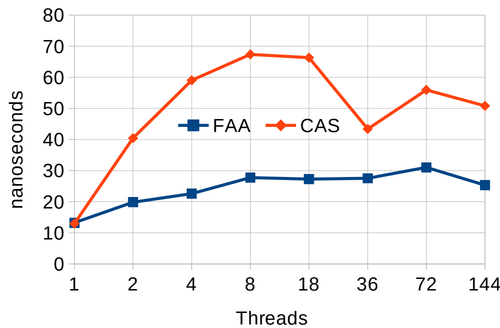


Figure 1 FAA vs. CAS on 4x18 Xeon E7-8880.

```
1  int Tail = 0, Head = 0; // Queue's tail and head
2  void * Array[∞];            // An infinite array
3  void enqueue(void * p)
4      while True do
5          T = FAA(&Tail, 1);
           // Repeat the loop if the entry is
           // already invalidated by dequeue()
6          if ( SWAP(&Array[T], p) = ⊥ )
7              return;

8  void * dequeue()
9      while True do
10         H = FAA(&Head, 1);
11         p = SWAP(&Array[H], ⊤);
12         if ( p ≠ ⊥ ) return p;
13         if ( Load(&Tail) ≤ H + 1 )
14             return nullptr;          // Empty
```

Figure 2 Infinite array queue (susceptible to livelocks).

FAA may seem to be a natural fit for ring buffers when updating queue's head and tail pointers. The reality, however, is more nuanced when designing lock-free queues. Many straight-forward algorithms without explicit locks that use FAA, e.g., [10], are actually not lock-free [4] because it is possible to find an execution pattern where no thread can make progress. Lack of true lock-freedom manifests in suboptimal performance when some threads are preempted since other threads are effectively blocked when the preemption happens in the middle of a queue operation. Additionally, such queues cannot be safely used in environments where blocking is not permitted. Even if FAA is not used, certain queues [23] fail to achieve linearizable lock-free behavior. A case in point: an open source lock-free data structure library, liblfds [12], simply falls back [13] to the widely known Michael & Scott's (M&S) FIFO lock-free queue [16] in their ring buffer implementation. While easy to implement, this queue does not scale well as we show in Section 7.

Despite the aforementioned challenges, the use of FAA is re-invigorated by recent concurrent FIFO queue designs [24, 19]. Unfortunately, [24, 19], despite their good performance, are not always memory efficient as we demonstrate in Section 7. Furthermore, such queues rely on memory allocators and safe memory reclamation schemes, thus creating a "chicken and egg" situation when allocating memory blocks. For example, if we simply want to recycle memory blocks or create a queue-based memory pool for allocation, reliance on an external memory allocator to allocate and deallocate memory is undesirable. Furthermore, typical system memory allocators, including jemalloc [2], are not lock-free. Lock-based allocators can defeat the purpose of creating a purely lock-free algorithm since they weaken overall progress guarantees. Moreover, in a number of use cases, such as within OS kernels, blocking can be prohibited or undesirable.

Contributions of the paper

- We introduce an approach of building ring buffers using indirection and two queues.
- We present our *scalable circular queue* (SCQ) design which uses FAA. To the best of our knowledge, it is the first ABA-safe design that is scalable, livelock-free and relies only on single-width atomic operations. It is inspired by CRQ [19] but prevents livelocks and uses our indirection approach. Although CRQ attempts to solve a similar problem,

it is livelock-prone, uses double-width CAS (unavailable on PowerPC [7], MIPS [17], RISC-V [21], SPARC [20], and other architectures) and is not standalone; it uses M&S queue as an extra layer (LCRQ) to work around livelock situations.

- Since unbounded queues are also used widely, we present the LSCQ design (Section 5.3) which chains SCQ ring buffers in a list. LSCQ is more memory efficient than LCRQ.

2 Background

Lock-free algorithms

We consider an algorithm lock-free if at least one thread can make progress in a finite number of steps. In other words, individual threads may starve, but a preempted thread must not block other threads from making progress. In contrast, spin locks (either implicit, found in some incorrect algorithms, or explicit) will prevent other threads from making further progress if the thread holding the lock is scheduled out.

Atomic primitives

CAS (compare-and-set) is used universally by most lock-free algorithms. However, one downside of CAS is that it can fail, especially under large contention. Although specialized instructions such as FAA (fetch-and-add) and SWAP do not reduce memory contention directly, they are more efficiently implemented by hardware and never fail. FAA and SWAP are currently implemented by x86-64 [8], ARMv8.1+ [1], and RISC-V [21].

Safe memory reclamation (SMR)

Most non-trivial lock-free algorithms, including queues, require special treatment of memory blocks that need to be deallocated, as concurrent threads may still access memory referred to by pointers retrieved prior to the change in a corresponding lock-free data structure. For programming languages such as C/C++, where unmanaged code is prevalent, lock-free *safe memory reclamation* techniques such as hazard pointers [15] are used. The main high-level idea is that each accessed pointer must be protected by a corresponding API call. When done, the thread's pointer reservation can be reset. When SMR knows that a memory block can be returned safely to the OS, it triggers memory deallocation. Bounded SCQ does not need SMR, but certain other lock-free queues such as LCRQ rely on SMR by their design.

Infinite array queue

Figure 2 shows an infinite array queue, originally described for the LCRQ design [19]. This queue is susceptible to livelocks, but our infinite array queue as well as the SCQ design are inspired by it. Initially, the queue is empty, i.e., all its entries are set to a special $\perp$ value. *enqueue* uses FAA on `Tail` to retrieve a slot which will be used to place a new entry. An enqueuer will try to use this slot. However, if the previous value is not $\perp$, some dequeuer already modified it, and this enqueuer moves on to the next slot. *dequeue* uses FAA on `Head` to retrieve a slot which contains a previously produced entry. A dequeuer will insert another special value, $\top$, to indicate that the slot can no longer be used. If the previous value is not $\perp$, a corresponding enqueuer has already produced some entry, which is taken by this dequeuer. Otherwise, this dequeuer moves on to the next slot. A corresponding enqueuer, which arrives later, will be unable to use this slot.

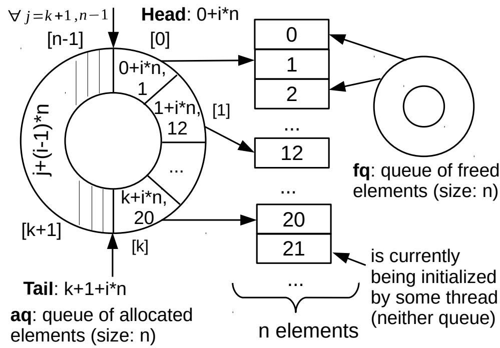


Figure 3 Proposed data structure.

```
   // data:  an array of pointers
   // aq is initialized empty
   // fq is initialized full
1  bool enqueue_ptr(void * ptr)
2      int index = fq.dequeue();
3      if ( index = ∅ ) return False;        // Full
4      data[index] = ptr;
5      aq.enqueue(index);
6      return True;                          // Success
7  void * dequeue_ptr()
8      int index = aq.dequeue();
9      if ( index = ∅ ) return nullptr;      // Empty
10     ptr = data[index];
11     fq.enqueue(index);
12     return ptr;                           // Success
```

Figure 4 Example: storing pointers.

3 Preliminaries

Assumptions

We assume that a program has k threads that can run on any number of physical CPU cores. For the purpose of this work, we will assume that the maximum (bounded) queue size is n. As no thread should block on another thread, we will further reasonably assume that $k \leq n$.

For simplicity of our presentation, we will assume that the system memory model is sequentially consistent [11]. However, the actual implementations of the algorithms (including implementations used in Section 7) can rely on weaker memory models whenever possible.

Data structure

Our design is based on two key ideas. First, we use indirection, i.e., data entries are not stored in the queue itself. Instead, a queue entry simply records an index into the array of data. Second, we maintain two queues: **fq**, which keeps indices to unallocated entries of the array, and **aq**, which keeps allocated indices to be consumed. A producer thread dequeues an index from **fq**, writes data to the corresponding array entry, and inserts the index into **aq**. A consumer thread dequeues the index from **aq**, reads data from the array, and inserts the entry back into **fq**.

Both queues maintain `Head` and `Tail` references (Figure 3). They are incremented when new entries are enqueued (`Tail`) or dequeued (`Head`). At any point, these references can be represented as $j + i \times n$, where j is an *index* (position in the ring buffer), i is a *cycle*, and n is a ring buffer size (must be power of 2 in our implementation). For example, for `Head`, we can calculate index $j = (Head \mod n)$ and cycle $i = (Head \div n)$.

Queue entries mirror `Head` and `Tail` values. Each entry also records an index into the array, pointing to the data associated with the entry. Unlike `Head` and `Tail`, it suffices to just record *cycle* i, as entry positions are redundant. We instead record an index into the array that is of the same bit-length as the position.

ABA safety

The ABA problem is prevented by comparing cycles. As both `Head` and `Tail` are incremented sequentially, regardless of queue size, they will not wrap around until after the number of operations exceeds the CPU word's largest value, a reasonable assumption made by other ABA-safe designs as well.

```
1  int Tail = n, Head = n;                // Initialization
2  forall entry_t Ent ∈ Entries[n] do
3    Ent = { .Cycle: 0, .Index: 0 };
4  void enqueue(int index)
5    do
6      T = Load(&Tail);
7      j = Cache_Remap(T mod n);
8      Ent = Load(&Entries[j]);
9      if ( Cycle(Ent) = Cycle(T) )
10       CAS(&Tail, T, T + 1);        // Help to
11       goto 6;                      // move tail
12     if ( Cycle(Ent) + 1 ≠ Cycle(T) )
13       goto 6;          // T is already stale
14     New = { Cycle(T), index };
15   while !CAS(&Entries[j], Ent, New);
16   CAS(&Tail, T, T+1);      // Try to move tail
```

```
17 int dequeue()
18   do
19     H = Load(&Head);
20     j = Cache_Remap(H mod n);
21     Ent = Load(&Entries[j]);
22     if ( Cycle(Ent) ≠ Cycle(H) )
23       if ( Cycle(Ent) + 1 = Cycle(H) )
24         return ∅;        // Empty queue
25       goto 19;       // H is already stale
26   while !CAS(&Head, H, H+1);
27   return Index(Ent);
```

Figure 5 Naive circular queue (NCQ).

Data entries and pointers

Data array entries can be of any type and size. It is not uncommon for programs to use a pair of queues with recyclable elements, e.g., queues analogous to **aq** and **fq**. In this case, a program can simply use indices instead of pointers. It is also possible to simply store (arbitrary) pointers as data entries. We can build a FIFO queue with data pointers using **aq** and **fq** queues as shown in Figure 4. A producer thread dequeues an entry from **fq**, initializes it with a pointer and inserts the entry into **aq**. A consumer thread dequeues the entry from **aq**, reads the pointer and inserts the entry back into **fq**. Note that *enqueue* does not need to check if a queue is full. It is only called when an available entry (out of n) exists (e.g., can be dequeued from **fq** if enqueueing to **aq**, or vice versa).

4 Naive Circular Queue (NCQ)

Let us first consider a simple algorithm, NCQ, which uses the presented data structure but borrows an idea of moving queue's tail on behalf of another thread from M&S queue [16]. It achieves performance similar to M&S queue but does not need double-width CAS to avoid the ABA problem. We use this queue as an extra baseline in Section 7.

Figure 5 shows the *enqueue* and *dequeue* operations. In the algorithm, we assume ordinary unsigned integer ring arithmetic when calculating cycles. Empty queues initialize all entries to cycle 0. Their `Head` and `Tail` are both n (cycle 1). Full queues initialize all entries to cycle 0 along with allocated entry indices. Their `Head` is 0 (cycle 0) and `Tail` is n (cycle 1).

Entries are always updated sequentially. To reduce contention due to false sharing, we remap queue entry positions by using a simple permutation function, *Cache_Remap*, that places two adjacent entries into different cache lines. The function remaps entries such that the same cache line will not be reused in the ring buffer as long as possible.

When dequeuing, we verify that `Head`'s cycle number matches the cycle number of the entry `Head` is pointing to. If `Head` is one cycle ahead, the queue is empty. Any other mismatches (i.e., an entry is ahead) imply that a producer has recycled this entry already. Therefore, other threads must have already consumed the entry and incremented `Head` since then (i.e., the previously loaded `Head` value is stale).

As discussed in Section 3, enqueueing is only possible when an available entry exists. To successfully enqueue an entry, `Tail` must be one cycle ahead of an entry it currently points to. `Tail`'s cycle number equals the entry's cycle number if another thread has already

```
   // Threshold prevents livelocks
1  signed int Threshold = -1;            // Empty queue
2  void enqueue(int index)
3      while True do
4          T = FAA(&Tail, 1);
5          if ( SWAP(&Entries[T], index) = ⊥ )
6              Store(&Threshold, 2n − 1);
7              return;

8  int dequeue()
9      if ( Load(&Threshold) < 0 ) return ∅;
10     while True do
11         H = FAA(&Head, 1);
12         index = SWAP(&Entries[H], ⊤);
13         if ( index ≠ ⊥ ) return index;
14         if ( FAA(&Threshold, -1) ≤ 0 )
15             return ∅
16         if ( Load(&Tail) ≤ H + 1 ) return ∅;
```

Figure 6 Infinite array queue. (We make it livelock-free by using a “threshold”.)

inserted an element but has not yet advanced `Tail`. The current thread helps to advance `Tail` to facilitate global progress. All other cycle mismatches (i.e., an entry is ahead) imply that the fetched `Tail` value is stale, as there has been at least an entire round (n enqueues) since it was last loaded.

5 Scalable Circular Queue (SCQ)

We will now consider a more elaborated design. Our scalable circular queue (SCQ) is partially inspired by CRQ [19] as well as by our data structure (Section 3). The major problem with CRQ is that it is not standalone due to its inherent susceptibility to livelocks. CRQ must be coupled with a truly lock-free FIFO queue (such as M&S queue [16]). If a livelock happens while enqueueing entries, a slow path is taken, where the current CRQ instance is “closed”. Then a new CRQ instance is allocated and used to enqueue new entries. This approach replaces CRQ with a list of CRQs (LCRQ). As discussed in introduction, this design has to rely on a memory allocator and memory reclamation scheme.

SCQ is not only standalone and livelock-free, but also much more portable across different CPU architectures. Unlike CRQ that requires a special double-width CAS instruction, our SCQ algorithm only needs single-width CAS, available across virtually all modern architectures. SCQ enables support for PowerPC [7] where CRQ/LCRQ cannot be implemented [24, 19]. Similarly, SCQ enables support for MIPS [17], SPARC [20], and RISC-V [21] which, like PowerPC, do not support double-width CAS.

5.1 Infinite array queue

We start off with the presentation of our infinite array queue. We diverge from the original idea (Section 2) in two major ways.

First, we present a solution to livelocks caused by dequeuers by introducing a special “threshold” value that we describe below. Livelocks occur when dequeuers incessantly invalidate slots that enqueuers are about to use for their new entries. By using the threshold value, we do not carry over the livelock problem to the practical implementation as in case of CRQ, i.e., guarantee that at least one enqueuer as well as one dequeuer both succeed after a finite number of steps at any point of time. Algorithms with this property were previously called *operation-wise* lock-free [19]. This term represents a stronger version of lock-freedom.

Second, our queue reflects the design presented in Section 3, where we use indices into the array of data rather than pointers. This approach guarantees that *enqueue* always succeeds (neither **aq** nor **fq** ever ends up with more than n elements). Rather, both “full” and “empty” conditions are detected by the corresponding *dequeue* operation as in Figure 4. Consequently, *enqueue* does not need to be treated specially to detect full queues.

We note that a circular queue accommodates only a finite number of elements, n. Furthermore, per assumptions in Section 3, the number of concurrent enqueuers or dequeuers never exceeds n ($k \leq n$). We apply these restrictions to our infinite queue and present a modified algorithm in Figure 6.

Suppose that *enqueue* successfully inserts some entry into the queue and sets the threshold on Line 6 prior to completion. Let us consider the very last inserted entry for which the threshold is set. (The threshold can also be set by any concurrent *enqueue* for preceding entries if their Line 6 executes after last enqueuer's Line 5.) We will justify the threshold value later. If dequeuers are active at that moment, we have either of the two scenarios:

The last dequeuer is not ahead of the inserted entry. In this case, the last dequeuer is no farther than n slots to the left of the inserted entry (Figure 7). This is because we have at most n available slots which can be referenced by any concurrent enqueuers. None of the enqueuers in this region (i.e., after the last dequeuer) can fail because the corresponding slots are not being invalidated by the dequeuers. (Note that this argument is only applicable to the infinite array queue. We will further refine it for SCQ below.)

The last dequeuer gets ahead of the inserted entry (either initially, or in the process of dequeuing). Any preceding concurrent dequeuer either succeeds when its entry can be consumed (Line 13), or fails and retries. Since `Head` increases monotonically (Line 11), if a failed dequeuer ever retries, its new position can only be after the current position of the last dequeuer. However, since the inserted entry is the very last for which *enqueue* is complete (Line 6), all of the dequeuers located after the last dequeuer (inclusively) are doomed to fail unless some other concurrent enqueuer completes. (In the latter case, we recursively go back to the very beginning of our argument where we have chosen the last inserted entry.)

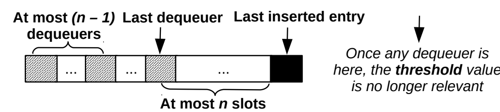


Figure 7 Threshold bound for livelock prevention.

In the second scenario, we are not concerned about the threshold value. In other words, Lines 14-15 that terminate dequeuers after the inserted entry do not cause any problems. All these dequeuers are guaranteed to fail one way or the other. Some preceding dequeuer, which is already in progress, will eventually fetch the inserted entry. However, in the first scenario, we want to make sure that Line 14 does not prematurely terminate dequeuers that are still trying to reach the inserted entry. Specifically, any failed attempt is penalized by decreasing the threshold value (Line 14).

To reach the last inserted entry, the last dequeuer, or any dequeuer that follows it later, will unsuccessfully traverse at most n slots (Figure 7). At the moment when the entry is inserted, we may also have up to $n-1$ dequeuers (since $k \leq n$) that are lagging behind (any distance away). When they fail, they are penalized by subtracting the threshold value. When they retry, the are guaranteed to be after the last dequeuer (Line 11). Thus, to guarantee that a dequeuer eventually reaches the inserted entry, the threshold must be $2n-1$.

Note that the threshold approach carefully avoids memory contention on the fast path in *dequeue*. Only *enqueue* typically updates this value through an ordinary memory write with a barrier. Moreover, if the threshold is still intact, it does not need to be updated.

```
1  int Tail = 2n, Head = 2n;                    // Empty queue
2  signed int Threshold = -1;
3  forall entry_t Ent ∈ Entries[2n] do
4      Ent = { .Cycle=0, .IsSafe=1, .Index=⊥ };
5  void catchup(int tail, int head)             // Internal
6      while !CAS(&Tail, tail, head) do
7          head = Load(&Head);
8          tail = Load(&Tail);
9          if ( tail ≥ head )
10             break;
11 void enqueue(int index)
12     while True do
13         T = FAA(&Tail, 1);
14         j = Cache_Remap(T mod 2n);
15         Ent = Load(&Entries[j]);
16         if ( Cycle(Ent) < Cycle(T) and
             Index(Ent) = ⊥ and
             (IsSafe(Ent) or Load(&Head) ≤ T) )
17             New = { Cycle(T), 1, index };
18             if ( !CAS(&Entries[j], Ent, New) )
19                 goto 15
20             if ( Load(&Threshold) ≠ 3n − 1 )
21                 Store(&Threshold, 3n − 1)
22             return;
23 int dequeue()
24     if ( Load(&Threshold) < 0 )      // Check if
25         return ∅;                    // the queue is empty
26     while True do
27         H = FAA(&Head, 1);
28         j = Cache_Remap(H mod 2n);
29         Ent = Load(&Entries[j]);
30         if ( Cycle(Ent) = Cycle(H) )
               // Cycle can't change, mark as ⊥
31             Atomic_OR(&Entries[j], { 0, 0, ⊥ });
32             return Index(Ent);            // Done
33         New = { Cycle(Ent), 0, Index(Ent) };
34         if ( Index(Ent) = ⊥ )
35             New = { Cycle(H), IsSafe(Ent), ⊥};
36         if ( Cycle(Ent) < Cycle(H) )
37             if ( !CAS(&Entries[j], Ent, New) )
38                 goto 29
39         T = Load(&Tail);                  // Check if
40         if ( T ≤ H + 1 ) // the queue is empty
41             catchup(T, H + 1);
42             FAA(&Threshold, -1);
43             return ∅;
44         if ( FAA(&Threshold, -1) ≤ 0 )
45             return ∅
```

Figure 8 Scalable circular queue (SCQ).

5.2 SCQ algorithm

In Figure 8, we present our SCQ algorithm. It is based on the modified infinite array queue and (partially) CRQ [19]. It manages cycles differently in *dequeue*, making it possible to leverage a simpler atomic OR operation instead of CAS.

Every entry in the SCQ buffer consists of the *Cycle* and *Index* components. We also reserve one bit in each entry, *IsSafe*, that we describe below. This bit is similar to the corresponding bit in CRQ.

SCQ leverages the high-level idea from the infinite array queue described above. However, SCQ replaces SWAP operations with CAS, as memory buffers are finite, and the same slot can be referenced by multiple cycles.

When discussing the threshold value, we previously assumed that no enqueuer can fail when all dequeuers are behind. In SCQ, this is no longer true, as the same entry can be occupied by some previous cycle. To bound the maximum distance between the last dequeuer and the last enqueuer, we ***double the capacity of the queue*** while still keeping the original number of elements. When using this approach, all the enqueuers after the last dequeuer can always locate an unused (⊥) slot no farther than $2n$ slots away from it. The threshold value should now become $(n - 1 + 2n) = 3n - 1$.

In the algorithm, we need a special value for ⊥. We reserve the very last index, $2n - 1$, for this purpose. Since, in SCQ, n is a power of 2 number, ⊥ will have all its index bits set to 1. As we show below, this allows *dequeue* to consume entries using an atomic OR operation. This value does not overlap with the actual data indices, which are still less than n.

SCQ also accounts for additional corner cases that are not present in the infinite queue:

A dequeuer arrives prior to its enqueuer counterpart, but the corresponding entry is already occupied. This happens when the entry is occupied by some other cycle. If this cycle is already newer (Line 36 is false), *dequeue* simply retries because the enqueuer counterpart is guaranteed to fail (Line 16). However, if the cycle is older, *dequeue* needs to mark it accordingly, so that when the enqueuer counterpart arrives, it will fail. For

```
1  void * ListHead = <empty SCQ>;
2  void * ListTail = ListHead;
3  void finalize_SCQ(SCQ * cq)
4      Atomic_OR(&cq.Tail, {.Value=0, .Finalize=1});
5  void * dequeue_unbounded()
6      while True do
7          SCQ * cq = Load(&ListHead);
8          void * p = cq.dequeue_ptr();
9          if ( p ≠ nullptr ) return p;
10         if ( cq.next = nullptr ) return nullptr;
11         Store(&cq.aq.Threshold, 3n − 1);
12         p = cq.dequeue_ptr();
13         if ( p ≠ nullptr ) return p;
14         if ( CAS(&ListHead, cq, cq.next) )
15             free_SCQ(cq);        // Dispose of cq
16 void enqueue_unbounded(void * p)
17     while True do
18         SCQ * cq = Load(&ListTail);
19         if ( cq.next ≠ nullptr )
20             CAS(&ListTail, cq, cq.next);
21             continue;            // Move list tail
           // Finalizes & returns false if full
22         if ( cq.enqueue_ptr(p, finalize=True) )
23             return;
24         ncq = alloc_SCQ();       // Allocate ncq
25         ncq.init_SCQ(p);  // Initialize & put p
26         if ( CAS(&cq.next, nullptr, ncq) )
27             CAS(&ListTail, cq, ncq);
28             return;
29         free_SCQ(ncq);           // Dispose of ncq
```

Figure 9 Unbounded SCQ-based queue (LSCQ).

this purpose, we clear the *IsSafe* bit, as in CRQ. The key idea is that the enqueuer will have to additionally make sure that all active dequeuers are behind when *IsSafe* is set to 0 (Line 16) before inserting a new entry. Whenever *IsSafe* becomes 0, only enqueuers can change it back to 1. (Only Line 17 sets the bit to 1; Line 35 preserves bit's value, and Line 33 sets it to 0.)

Enqueuers attempt to use slots that are marked unsafe. If *IsSafe* on Line 16 is 0, an enqueuer will additionally make sure that the dequeuer that needs to be accounted for has not yet started by reading and comparing the current `Head` value.

When dequeuing elements, if cycles match (Line 30), a dequeuer is guaranteed to succeed. The corresponding slot will not be recycled until an entry is consumed. The only thing that can change is the *IsSafe* bit. Unlike CRQ, to mark an entry as consumed, *dequeue* issues an atomic OR operation which sets all index bits to 1 while preserving entry's safe bit and cycle.

The *catchup* procedure is similar to the *fixState* procedure from CRQ and is used when the tail is behind the head. This allows to avoid unnecessary iterations in *enqueue* and reduces the risk of contention.

Finally, when comparing cycles, we use a common approach with signed integer arithmetic which takes care of potential wraparounds.

Optimization

Similarly to LCRQ, SCQ employs an additional optimization on dequeuers. If a dequeuer arrives prior to the corresponding enqueuer, it will not aggressively invalidate a slot. Instead, it will spin for a small number of iterations with the expectation that the enqueuer arrives soon. This alleviates unnecessary contention on the head and tail pointers, and consequently helps both dequeuers and enqueuers.

5.3 SCQ-based unbounded queue (LSCQ)

We follow LCRQ's main idea of maintaining a list of ring buffers in our LSCQ design. LSCQ is potentially more memory efficient than LCRQ, as it is based on livelock-free SCQs which do not end up being prematurely "closed" (Section 7). Since operations on the list are very rare, the cost is completely dominated by SCQ operations.

In Figure 9, we present the LSCQ algorithm. We intentionally ignore the memory reclamation problem (Section 2) which can be straight-forwardly solved by the corresponding techniques such as hazard pointers [15]. The presented unbounded queue can store any

```
1 bool enqueue_ptr(void * ptr)
      // Add a full queue check before Line 12:
2     T = Load(&Tail);
3     if ( T ≥ Load(&Head) + 2n ) return False;
4     ... Modified enqueue() ...
      // Add a full queue check in the loop after Line 22:
5     if ( T+1 ≥ Load(&Head) + 2n ) return False;
```

Figure 10 SCQ for double-width CAS: checking for full queues.

fixed-size data entries, including pointers (as in Figure 9), just like SCQ itself. When a ring buffer is full, we need to additionally "finalize" it, so that no new entries are inserted by the concurrent threads. As in CRQ, we reserve one bit in `Tail`. When **fq** does not have available entries (Line 3, Figure 4), we set the corresponding bit for **aq**'s `Tail`.

We also modify *enqueue* for **aq** such that it fails when FAA on `Tail` returns a value with the "finalized" bit set. Thus, any concurrent thread that tries to insert entries after **aq** is being finalized, fails. In this case, we also need to place the entry back into **fq**. This cannot fail since **fq** is never finalized.

Before unlinking *cq* from the list (Line 14), *dequeue_unbounded* checks again that *cq*, which must already be finalized, is empty. Pending enqueuers may still access it. For the final check, the threshold must be reset so that slots for the pending enqueuers can be invalidated.

5.4 SCQ for double-width CAS

The x86-64 [8] and ARM64 [1] architectures implement double-width CAS, which atomically updates two contiguous words. We can leverage this capability to build SCQ which avoids indirection when storing arbitrary pointers. In this case, all entries consist of tuples. Each tuple stores two adjacent integers instead of just one integer. The *index* field of the first integer now simply indicates if the entry is occupied (0) or available ($\perp$). The second integer from the same tuple stores a pointer which is used in lieu of an index. Since pointers also need to be stored and retrieved, we change Lines 18, 31, and 37 to use double-width CAS accordingly.

This version of SCQ provides a fully compatible API such that architectures without double-width CAS can still implement the same queue through indirection. In this queue, *enqueue* becomes *enqueue_ptr*, and *dequeue* becomes *dequeue_ptr*.

If *enqueue_ptr* needs to identify full queues, additional changes are required. In Figure 10, we show a method which compares `Head` and `Tail` values. The comparison is relaxed and up to k ($k \leq n$) concurrent enqueuers can increment `Tail` spuriously. Thus, `Tail` can now be up to $3n$ slots ahead of `Head`. Since we previously assumed that number to be $2n$, we increase the threshold from $3n - 1$ to $4n - 1$. This method is imprecise and can only guarantee that at least n elements are in the queue, but the actual number varies. This is often acceptable, especially when creating unbounded queues (Section 5.3), which finalize full queues.

6 Correctness

We omit more formal linearizability arguments for NCQ due to its simplicity. SCQ's linearizability follows from the arguments we make in Sections 5.1 and 5.2, as well as from the corresponding CRQ linearizability derivations [19] because the SCQ design has many similarities with CRQ. Below we provide lock-freedom arguments for NCQ and SCQ.

▶ **Theorem 1.** *The NCQ algorithm is lock-free.*

Proof. The NCQ algorithm has two unbounded loops: one in *enqueue* (Lines 5-15) and the other one in *dequeue* (Lines 18-26).

If CAS fails in *enqueue* causing it to repeat, it means that the corresponding entry `Entries[j]` is changed by another thread executing *enqueue*, as *dequeue* does not modify entries. Consequently, that other thread is making progress, i.e., succeeding in the *enqueue* operation (Line 15).

If CAS fails in *dequeue* causing it to repeat, it means that the `Head` pointer is modified by another thread executing *dequeue*. Therefore, that other thread is making progress, i.e., succeeding in the *dequeue* operation (Line 26). ◀

▶ **Theorem 2.** *The SCQ algorithm is lock-free.*

Proof. The SCQ algorithm has two unbounded loops: one in *enqueue* (Lines 12-22) and the other one in *dequeue* (Lines 26-45).

If the condition on Line 16 of *enqueue* is false causing it to repeat the loop, then two possibilities exist. First, some dequeuer already invalidated the slot for this enqueuer (*Cycle(Ent)* and *IsSafe(Ent)* checks) because it arrived before the enqueuer. Alternatively, the entry is occupied by a prior enqueuer (i.e., $\neq \perp$) and is supposed to be consumed by some dequeuer which is yet to come.

In the latter case, the current enqueuer skips the occupied slot. There may also exist other concurrent enqueuers which will skip occupied slots as well. Since the total number of elements for *enqueue* is always capped, the queue will never have more than n entries. Thus, the enqueuers should be able to succeed unless dequeuers keep invalidating their slots.

The first case is more intricate. If none of the enqueuers succeed, dequeuers must not be able to invalidate new slots after a finite number of steps. Once dequeuers stop invalidating slots (using the threshold described in Section 5.1), at least one enqueuer can make further progress by catching up its `Tail` to the next available position and inserting a new element.

If the conditions on Lines 30, 40, or 44 of *dequeue* are false causing it to repeat the loop, then the following must be the reason for that. Line 30 can only be false if the entry is not yet initialized by the corresponding enqueuer. In this case, the dequeuer may potentially iterate and invalidate slots as many as $3n$ times until either Line 43 or 45 terminates the loop. Line 45 is guaranteed to eventually terminate the loop as long as no new entries are inserted by *enqueue* (i.e., enqueuers can be running, but none of them succeed). Any pending (almost completed) enqueuer may still cause Line 21 to increase the threshold value temporarily even though its entry was already consumed. However, this at most is going to happen for $k-1$ pending enqueuers. After that, the threshold value will be depleted causing all active dequeuers to complete (Line 45). Since dequeuers are no longer running, at least one new enqueuer will be able to succeed. At that point, it will reset the threshold value (Line 21). After that, we recursively repeat this entire argument again to show that at least one following enqueuer will succeed in a finite number of steps. ◀

7 Evaluation

In this section, we evaluate our SCQ design against well-known or state-of-the-art algorithms. We use and extend the benchmark from [24] which already implements several algorithms.

In the evaluation, we show that SCQ achieves very high performance while avoiding limitations that are typical to other high-performant algorithms (i.e., livelock workarounds, memory reclamation, and portability). We have also found that state-of-the-art approaches, especially LCRQ, can have very high memory usage, a problem that does not exist in SCQ.

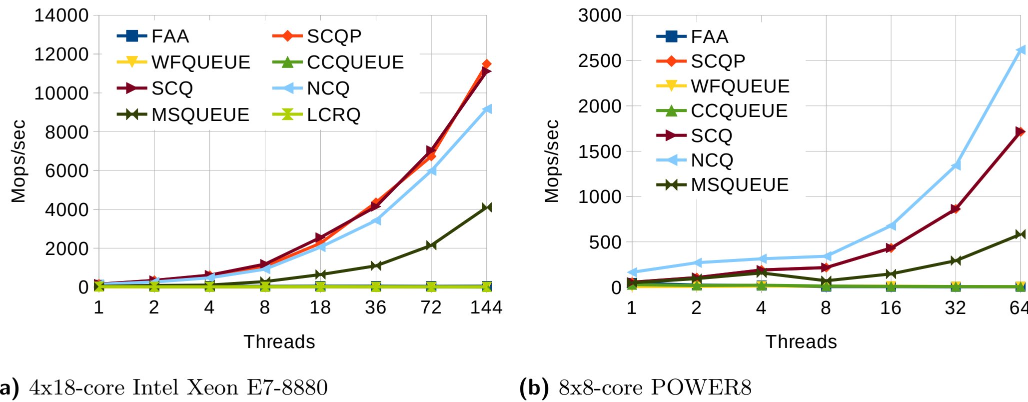
We present results for both bare-bones SCQ and the version that stores arbitrary pointers (SCQP). Bare-bones SCQ is relevant because queue elements in SCQ can be of any type, i.e., not necessarily pointers. We compare SCQ and SCQP against M&S FIFO lock-free queue (MSQUEUE) [16], combining queue (CCQUEUE) [3] – which is not a lock-free queue but is known to have good performance, LCRQ [19] – a queue that maintains a lock-free list of ring buffers (CRQ); CRQs cannot be used separately, as they are susceptible to livelocks, WFQUEUE – a recent scalable wait-free queue design [24]. These algorithms provide reasonable baselines as they represent well-known or state-of-the-art (in scalability) approaches. We also present NCQ as an additional baseline. NCQ uses the same data structure as SCQ, but its design is reminiscent of MSQUEUE. Finally, we provide FAA (fetch-and-add) throughputs to show the potential for scalability. FAA is not a real algorithm, it simply implements atomic increments on `Head` and `Tail` when calling *dequeue* and *enqueue* respectively. We skip a separate LSCQ evaluation since SCQ is already lock-free and can be used as is. LSCQ's costs are largely dominated by the underlying SCQ implementation.

Performance differences between queues should be treated with great caution. For example, LCRQ has better throughput in a few tests, but it consumes a lot of memory under certain circumstances. Also, neither LCRQ nor WFQUEUE are deployable in all places where SCQ can be used, e.g., fixed-size data pools. As mentioned in introduction, this would trigger a "chicken and egg" situation, where data pools would need to depend on some external (typically non lock-free) memory allocator. Moreover, both LCRQ and WFQUEUE require safe memory reclamation by their design; the benchmark implements a specialized scheme for WFQUEUE and hazard pointers [15] for LCRQ and MSQUEUE. Finally, WFQUEUE needs special per-thread descriptors, which reduce the API transparency.

We run all experiments for up to 144 threads on a 72-core machine consisting of four Intel Xeon E7-8880 v3 CPUs with hyper-threading disabled, each running at 2.30 GHz and with a 45MB L3 cache. The machine has 128GB of RAM and runs Ubuntu 16.04 LTS. We use gcc 8.3 with the -O3 optimization flag. For SCQP, we use the double-width version (Section 5.4) as x86-64 can benefit from it. SCQP is compiled with clang 7.0.1 (-O3) because it generates faster code for our implementation (no such advantage for other queues).

We also evaluate queues on the PowerPC architecture. We run experiments on an 8-core POWER8 machine. Each core has 8 threads, so we have 64 logical cores in total. We do not disable PowerPC's simultaneous multithreading because the machine does not have as many cores as our Xeon testbed. All cores are running at 3.0 Ghz and have an 8MB L3 cache. The machine has 64GB of RAM and runs Ubuntu 16.04 LTS. We use gcc 8.3 with the -O3 optimization flag. Since PowerPC does not support double-width CAS, it is impossible to implement the LCRQ algorithm there. In contrast, SCQ and SCQP both work well on PowerPC. For SCQP, we use the version with indirection and two queues.

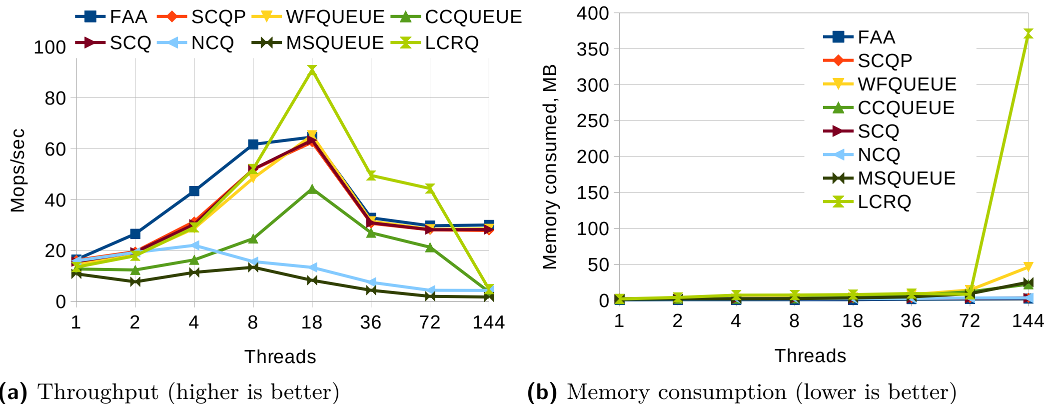
We use jemalloc [2] to alleviate libc's malloc poor performance [14]. Each data point is measured 10 times for 10000000 operations in a loop, we present the average. The benchmark measures throughput in a steady/hot state and protects against occasional outliers. We use the default benchmark parameters from [24] to achieve optimal performance with LCRQ, CCQUEUE, and WFQUEUE. For SCQ and NCQ, we have chosen a relatively small ring buffer size, 2^{16} entries. (SCQ uses only half queue's capacity, $n = 2^{15}$ entries, as discussed in Section 5.2.) LCRQ uses 2^{12} entries in each CRQ to attain optimal performance. Unlike SCQ, LCRQ wastes a lot of memory in each CRQ due to cache-line padding. Most of our results for x86-64 have peaks for 18 threads because each CPU has 18 cores. Over-socket contention is expensive and results in performance drops. PowerPC has an analogous picture.



(a) 4x18-core Intel Xeon E7-8880 **(b)** 8x8-core POWER8

Figure 11 Empty queue test, throughput of the dequeue operation.

In Figure 11, we perform a simple experiment to measure the cost of *dequeue* on empty queues. MSQUEUE, NCQ, SCQ, and SCQP perform reasonably well on both x86-64 (Figure 11a) and PowerPC (Figure 11b) since they do not dequeue elements aggressively. LCRQ, WFQUEUE, and CCQUEUE take a performance hit in this corner case. FAA is also slower than MSQUEUE, NCQ, SCQ, and SCQP because it still needs to modify an atomic variable. Slow dequeuing on empty queues was previously acknowledged by WFQUEUE's authors [24].

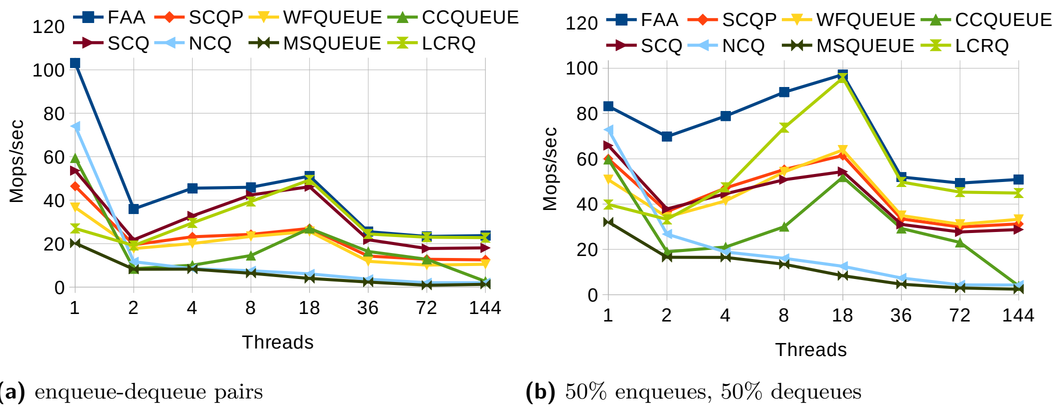


(a) Throughput (higher is better) **(b)** Memory consumption (lower is better)

Figure 12 Memory efficiency test, 4x18-core Intel Xeon E7-8880 (standard malloc).

To evaluate memory efficiency, we run an experiment with 50% of *enqueue* and 50% of *dequeue* operations that are chosen by each thread randomly (Figure 12). For this test, we use libc's standard malloc to make sure that memory pages are unmapped more aggressively. We run the benchmark with its default configuration that uses tiny delays between operations. Using delays allows us to get more pronounced results while still showing a realistic execution scenario. It turns out that while, for the most part, LCRQ provides higher throughput (Figure 12a), it can also allocate a lot of memory while running (Figure 12b), up to $\approx$ 400MB. WFQUEUE's memory usage is also somewhat elevated (up to $\approx$ 50MB) and exceeds that of MSQUEUE and CCQUEUE for most data points. Conversely, SCQ, SCQP, and NCQ are very efficient; they only need a small (512K-1MB), fixed-size buffer that is allocated for circular queues. Overall, SCQ and SCQP win here as they both achieve great performance with very little memory overhead.

Since the design of SCQ is related to LCRQ, we were particularly interested in investigating LCRQ's high memory usage. As we suspected, LCRQ was "closing" CRQs frequently due to livelocks. To maintain good performance, LCRQ must use relatively large CRQs (each of them has 2^{12} entries). However, due to livelocks, CRQs need to be prematurely closed from time to time. Eventually, LCRQ ends up in a situation where it frequently allocates new CRQs, i.e., wasting memory greatly. Safe memory reclamation additionally impacts the timing of deallocation, i.e., CRQs are not deallocated immediately.

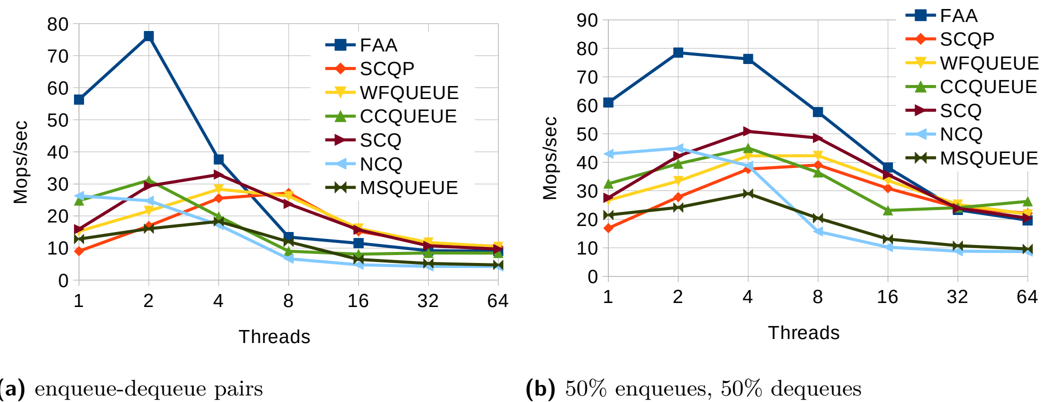


(a) enqueue-dequeue pairs

(b) 50% enqueues, 50% dequeues

Figure 13 Balanced load tests, 4x18-core Intel Xeon E7-8880.

Finally, we evaluate queues using operations by multiple threads in a tight loop. In Figures 13a and 14a, we present throughput for the x86-64 and PowerPC architectures respectively when using pairwise queue operations. In this experiment, every thread executes *enqueue* followed by *dequeue* in a tight loop. Since multiple concurrent threads are running simultaneously, the order of dequeued elements is not predetermined anyhow (even though enqueue and dequeue are paired). For x86-64, SCQ and LCRQ are both winners and have roughly similar performance. SCQP, WFQUEUE, and CCQUEUE (partially) attain half of their throughput on average. For PowerPC, SCQ is a winner: it generally outperforms all other algorithms. SCQP and WFQUEUE are roughly identical, except that WFQUEUE marginally outperforms SCQP for smaller concurrencies. CCQUEUE is generally worse, except very small concurrencies.



(a) enqueue-dequeue pairs

(b) 50% enqueues, 50% dequeues

Figure 14 Balanced load tests, 8x8-core POWER8.

In Figures 13b and 14b, we present results for an experiment which selects operations randomly: 50% of enqueues and 50% of dequeues. For x86-64, WFQUEUE and SCQP are almost identical. SCQP marginally outperforms SCQ when concurrency is high, probably due to (occasional) cache contention since entries are larger in double-width SCQP. CCQUEUE is typically slower than WFQUEUE, SCQP, or SCQ. LCRQ outperforms all of them most of the time. However, considering its memory utilization, LCRQ may not be appropriate in a number of cases as previously discussed. For PowerPC, SCQ is a winner: it generally outperforms all other algorithms. SCQP, WFQUEUE, and CCQUEUE are very close; WFQUEUE marginally outperforms SCQP in this test.

8 Related Work

Over the last couple of decades, different designs were proposed for concurrent FIFO queues as well as ring buffers. A classical Michael & Scott's lock-free FIFO queue [16] maintains a list of nodes. The list has the *head* and *tail* pointers. These pointers must be updated using CAS operations. The queue can be modified to avoid the ABA problem related to pointer updates. For that purpose, double-width CAS is used to store an ABA tag for each pointer.

Existing lock-free ring buffer designs that do not benefit from FAA (e.g., [22]) are typically not very scalable. Another approach [4], though uses FAA, needs a memory reclamation scheme and does not seem to scale as well as some other algorithms. Certain queues use FAA but are not linearizable. For example, [5] maintains a queue size and updates it with FAA. However, the queue may end up in inconsistent state as previously discussed in [19].

Certain bounded MPMC queues without explicit locks such as [10, 23] are relatively fast. However, these approaches are technically not lock-free as discussed in [4, 13]. Just as with explicit spin locks, lack of true lock-freedom manifests in suboptimal performance when threads are preempted, as remaining threads are effectively blocked when the preemption happens in the middle of a queue operation.

Alternative concurrent designs were also considered. CCQUEUE [3] is a special combining queue. The reasoning behind this queue is that it may be cheaper to execute operations sequentially. When performing an operation, a thread is added to a list; the thread at the head of the list completes operations on behalf of other threads from the list.

Finally, the use of FAA for queues is advocated by recent non-blocking FIFO queue designs [24, 19]. Unfortunately, [24, 19], despite their good performance, are not always memory efficient.

9 Conclusion

In this paper, we presented SCQ, a scalable lock-free FIFO queue. The main advantage of SCQ is that the queue is standalone, memory efficient, ABA safe, and scalable. At the same time, SCQ does not require a safe memory reclamation scheme or memory allocator. In fact, this queue itself can be leveraged for object allocation and reclamation, as in data pools.

We use FAA (fetch-and-add) for the most contended hot spots of the algorithm: `Head` and `Tail` pointers. Unlike prior attempts to build scalable queues with FAA such as CRQ, our queue is both lock-free and linearizable. SCQ prevents livelocks directly in the ring buffer itself without trying to work around the problem by allocating additional ring buffers and linking them together. SCQ is very portable and can be implemented virtually everywhere. It only needs single-width CAS. It is also possible to create unbounded queues based on SCQs which are more memory efficient than LCRQ.

References

1. Arm Limited. ARM Architecture Reference Manual. `http://developer.arm.com/`, 2019.
2. Jason Evans. A scalable concurrent malloc(3) implementation for FreeBSD. In *Proceedings of the BSDCan Conference, Ottawa, Canada*, 2006.
3. Panagiota Fatourou and Nikolaos D. Kallimanis. Revisiting the Combining Synchronization Technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 257–266, New York, NY, USA, 2012. ACM.
4. Steven Feldman and Damian Dechev. A Wait-free Multi-producer Multi-consumer Ring Buffer. *ACM SIGAPP Applied Computing Review*, 15(3):59–71, October 2015.
5. Eric Freudenthal and Allan Gottlieb. Process Coordination with Fetch-and-increment. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pages 260–268, 1991.
6. Danny Hendler, Nir Shavit, and Lena Yerushalmi. A Scalable Lock-free Stack Algorithm. In *Proceedings of the 16th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, pages 206–215, New York, NY, USA, 2004. ACM.
7. IBM. PowerPC Architecture Book, Version 2.02. Book I: PowerPC User Instruction Set Architecture. `http://www.ibm.com/developerworks/`, 2005.
8. Intel. Intel 64 and IA-32 Architectures Developer's Manual. `http://www.intel.com/`, 2019.
9. Christoph M. Kirsch, Michael Lippautz, and Hannes Payer. Fast and Scalable, Lock-Free k-FIFO Queues. In *Proceedings of the 12th International Conference on Parallel Computing Technologies - Volume 7979*, pages 208–223, Berlin, Heidelberg, 2013. Springer-Verlag.
10. Alexander Krizhanovsky. Lock-free Multi-producer Multi-consumer Queue on Ring Buffer. *Linux J.*, 2013(228), 2013. URL: `http://dl.acm.org/citation.cfm?id=2492102.2492106`.
11. Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.
12. Liblfds. Lock-free Data Structure Library. URL: `http://www.liblfds.org`.
13. Liblfds. Ringbuffer Disappointment. URL: `http://www.liblfds.org/wordpress/index.php/2016/04/29/ringbuffer-disappointment/`.
14. Lockless Inc. Memory Allocator Benchmarks. `https://locklessinc.com/benchmarks_allocator.shtml`, 2019.
15. Maged M. Michael. Hazard pointers: safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, June 2004.
16. Maged M. Michael and Michael L. Scott. Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, May 1998.
17. MIPS. MIPS32/MIPS64 Rev. 6.06. `http://www.mips.com/products/architectures/`, 2019.
18. Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit. Using Elimination to Implement Scalable and Lock-free FIFO Queues. In *Proceedings of the 17th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '05, pages 253–262, 2005.
19. Adam Morrison and Yehuda Afek. Fast Concurrent Queues for x86 Processors. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 103–112, New York, NY, USA, 2013. ACM.
20. Oracle. SPARC Architecture 2011. `http://www.oracle.com/`, 2019.
21. RISC-V Foundation. RISC-V Books. `http://riscv.org/risc-v-books/`, 2019.
22. Philippas Tsigas and Yi Zhang. A Simple, Fast and Scalable Non-blocking Concurrent FIFO Queue for Shared Memory Multiprocessor Systems. In *Proceedings of the 13th ACM Symposium on Parallel Algorithms and Architectures*, SPAA '01, pages 134–143, 2001.
23. Dmitry Vyukov. Bounded MPMC queue. URL: `http://www.1024cores.net/home/lock-free-algorithms/queues/bounded-mpmc-queue`.
24. Chaoran Yang and John Mellor-Crummey. A Wait-free Queue As Fast As Fetch-and-add. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 16:1–16:13, New York, NY, USA, 2016. ACM.

Byzantine Approximate Agreement on Graphs

Thomas Nowak

Université Paris-Sud, France
Centre National de la Recherche Scientifique, Paris, France
thomas.nowak@lri.fr

Joel Rybicki

Institute of Science and Technology Austria, Klosterneuburg, Austria
joel.rybicki@ist.ac.at

Abstract

Consider a distributed system with n processors out of which f can be Byzantine faulty. In the approximate agreement task, each processor i receives an input value x_i and has to decide on an output value y_i such that

1. the output values are in the convex hull of the non-faulty processors' input values,
2. the output values are within distance d of each other.

Classically, the values are assumed to be from an m-dimensional Euclidean space, where $m \geq 1$.

In this work, we study the task in a discrete setting, where input values with some structure expressible as a graph. Namely, the input values are vertices of a finite graph G and the goal is to output vertices that are within distance d of each other in G, but still remain in the graph-induced convex hull of the input values. For $d = 0$, the task reduces to consensus and cannot be solved with a deterministic algorithm in an asynchronous system even with a single crash fault. For any $d \geq 1$, we show that the task is solvable in asynchronous systems when G is chordal and $n > (\omega + 1)f$, where ω is the clique number of G. In addition, we give the first Byzantine-tolerant algorithm for a variant of lattice agreement. For synchronous systems, we show tight resilience bounds for the exact variants of these and related tasks over a large class of combinatorial structures.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases consensus, approximate agreement, Byzantine faults, chordal graphs, lattice agreement

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.29

Related Version A full version of the paper is available at `https://arxiv.org/abs/1908.02743`.

Funding *Thomas Nowak*: Supported by the CNRS project PEPS DEMO and the Université Paris-Saclay project DEPEC MODE.
Joel Rybicki: This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 754411.

Acknowledgements We thank the anonymous reviewers for their helpful comments and Janne H. Korhonen for many discussions on this work. We also wish to thank the participants of the Helsinki Workshop on Theory of Distributed Computing 2018 and the Metastability workshop in Mainz 2018 for discussions that lead to the problem of approximate agreement on graphs.

1 Introduction

In a distributed system, processors often need to coordinate their actions by jointly making consistent decisions or collectively agreeing on some data. While distributed systems can be resilient to failures, the extent to which they do so varies dramatically depending on the underlying communication and timing model, the fault model, and the level of coordination required by the task at hand. Exploring this interplay is at the core of distributed computing.

© Thomas Nowak and Joel Rybicki;
licensed under Creative Commons License CC-BY

In this work, we investigate to which degree agreement can be reached in message-passing systems with Byzantine faults when (1) the set of input values has some discrete, combinatorial structure and (2) the set of output values must satisfy some structural closure property over the input values. We consider deterministic algorithms and assume a system with fully-connected point-to-point communication topology consisting of n processors out of which f may experience Byzantine failures, where the faulty processors may arbitrarily deviate from the protocol (e.g., crash, omit messages, or send malicious misinformation). We consider both asynchronous and synchronous systems. In the former, the processors do not have access to a shared global clock and sent messages may take arbitrarily long (but finite) time to be delivered. In the synchronous case, computation and communication proceeds in a lock-step fashion over discrete rounds.

1.1 Fault-tolerant distributed agreement tasks

Let P denote the set of n processors and $F \subseteq P$ some (unknown) set of faulty processors, where $|F| \leq f$. Many distributed agreement problems take the following form: Each processor $i \in P$ receives some input value $x_i \in V$, where V is the set of possible input values. The task is to have every non-faulty processor $i \in P \setminus F$ (irreversibly) decide on an output value $y_i \in V$ subject to some *agreement* and *validity* constraints. These constraints are commonly defined over the sets $X = \{x_i : i \in P \setminus F\}$ of input and $Y = \{y_i : i \in P \setminus F\}$ of output values of *non-faulty* processors. By choosing different constraints, one obtains different types of agreement problems.

1.1.1 Consensus and k-set agreement

Consensus is one of the most elementary problems in distributed computing [49]: all non-faulty processors should output a single value (agreement) that was the input of some non-faulty processor (validity). A natural generalisation of consensus is the k-set agreement problem [8], which is defined by the following constraints:

- agreement: $|Y| \leq k$ (all non-faulty processors decide on at most k values),
- validity: $Y \subseteq X$ (each decided value was an input of some non-faulty processor).

The special case $k = 1$ is the consensus problem and is known to be impossible to solve in an asynchronous setting even with $V = \{0, 1\}$ under a single crash fault using deterministic algorithms [28]. Analogously, k-set agreement cannot in general be solved in an asynchronous message-passing systems if there are $f \geq k$ crash faults [33, 6]. Note that for k-set agreement, it is natural to consider also other validity constraints [9].

1.1.2 Approximate agreement

While consensus and k-set agreement cannot in general be solved in an asynchronous system, it is however possible to obtain *approximate* agreement – in the sense that output values are close to each other – even in the presence of Byzantine faults. Formally, in the (multidimensional) approximate agreement problem, we are given $\varepsilon > 0$ and the set $V = \mathbb{R}^m$ of values forms an m-dimensional Euclidean space for some $m \geq 1$. The task is to satisfy

- agreement: $\text{dist}(y, y') \leq \varepsilon$ for any $y, y' \in Y$ (output values are within Euclidean distance ε),
- validity: the set Y is contained in the convex hull $\langle X \rangle$ of the set X of nonfaulty input values.

For an arbitrary $m \geq 1$, Mendes et al. [47] showed that under Byzantine faults the problem is solvable in asynchronous systems if and only if $n > (m + 2)f$ holds.

1.1.3 Lattice agreement

Lattice agreement is another well-studied relaxation of consensus with applications in renaming problems and obtaining atomic snapshots [3, 2, 23, 58]. In this problem, the set V of values forms a *semilattice* $\mathbb{L} = (V, \oplus)$, i.e., an idempotent commutative semigroup. The $\oplus$ operator defines a partial order $\leq$ over V defined as $u \leq v \iff u \oplus v = v$. The task is to decide on values that lie on a non-trivial chain, i.e., values that are comparable under $\leq$:

- agreement: $y \leq y'$ or $y' \leq y$ for any $y, y' \in Y$,
- validity: for any $y \in Y$ there exists some $x \in X$ such that $x \leq y$ and $y \leq \bigoplus X$.

Note that under crash faults the validity condition is usually given as $x_i \leq y_i \leq \bigoplus\{x_j : j \in P\}$ for $i \in P \setminus F$, which is less suitable in the context of Byzantine faults since otherwise output values could exit the convex hull defined by the correct processes' input values.

1.2 Structured agreement problems

Unlike the k-set agreement problem, the approximate and lattice agreement problems impose additional structure on the set V of values. In the former, the values form a (continuous) m-dimensional Euclidean space, whereas in the latter there is algebraic structure. Furthermore, the validity conditions require that the output respects some *closure property* on the input values. In approximate agreement, the closure is given by the convex hull operator in Euclidean spaces, whereas in lattice agreement, the output must reside in the minimal superset of X closed under $\oplus$.

Such closure systems have been studied under the notion of *abstract convexity spaces* and have a rich theory [36, 21, 18, 22, 55]. A (finite) convexity space on V is a collection $\mathcal{C}$ of subsets of V that satisfies

1. $\emptyset, V \in \mathcal{C}$,
2. $A, B \in \mathcal{C}$ implies $A \cap B \in \mathcal{C}$.

As the name suggests, the sets in $\mathcal{C}$ are called *convex* and every convexity space has the natural closure operator, which maps any set $A \in V$ to a minimal convex superset $A \subseteq \langle A \rangle \in \mathcal{C}$ called the convex hull of A. *Convex geometries* [21] are an important class of convexity spaces, which satisfy the Minkowski-Krein-Milman property: the closure $\langle A \rangle$ of any set $A \subseteq V$ is the closure of its *extreme points*, where $a \in A$ is an extreme point of A if $a \notin \langle A \setminus a \rangle$. Convex geometries have been studied extensively in a wide variety of *combinatorial* structures, such as graphs and hypergraphs [34, 24, 25, 19, 50, 48, 17], and partially ordered sets [18, 21, 51].

There has been extensive research on developing theory of convexity over *combinatorial* structures, such as graphs and hypergraphs [34, 24, 25, 19, 50, 48, 17], partially ordered sets [18, 21, 51], and so on. Much of the research has focused on identifying analogues to classical convexity invariants, such as Helly, Carathéodory, and Radon numbers, in various abstract convexity spaces [36, 34, 19, 20, 4, 17]. Convex geometries also have deep connections with matroid and antimatroid theory [11, 39]: convex geometries are duals of antimatroids, and a special class of greedoids, which provide a structural framework for characterising greedy algorithms [38, 39], are convex geometries [21]. Lovász and Saks [44] used theory of convex geometries to analyze a broad class of two-party communication complexity problems.

1.3 Approximate agreement on graphs

As our main example of an agreement problem with discrete, combinatorial structure, we focus on a problem where the set V of values has relational structure in the form of a connected graph $G = (V, E)$. In the *monophonic approximate agreement problem on* G the task is to output a set of vertices that satisfy

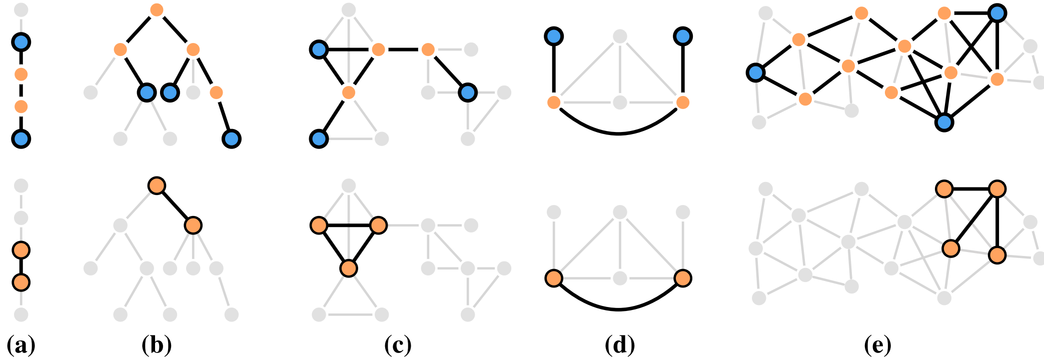


Figure 1 Examples of geodesic and monophonic agreement on graphs. In the top row, the blue and orange vertices form a convex hull of the blue vertices for each graph under (a)–(d) geodesic and (e) monophonic convexities. The thick edges lie in the shortest (geodesic) or chordless (monophonic) paths between the blue vertices. The bottom row shows possible feasible outputs for the respective approximate agreement problems with $d = 1$, i.e., the highlighted vertices form a clique (agreement) and are contained in the respective convex hull of the input values (validity).

- agreement: the set Y of output has diameter at most d for a given $d \geq 1$,
- validity: each value $y \in Y$ lies on a chordless[1] path between some input vertices $x, x' \in X$.

The above problem is a natural generalisation of approximate agreement onto graphs. It is easy to see that the discrete version of one-dimensional approximate agreement is just approximate agreement on a path (Figure 1a). If G is a tree or a block graph[2], then the task is to output vertices that lie on the minimal vertex set connecting all input vertices (Figure 1b–c).

In the parlance of abstract convexity theory [34, 24, 25, 19], the validity condition requires that the output lies in the *monophonic*, or *minimal path*, or *chordless path* convex hull of the input vertices. Another reasonable validity constraint would be to require the output values to lie on the *shortest* paths between input vertices, i.e., in the *geodesic* convex hull. We consider both variants and refer to the latter version of the problem as *geodesic* approximate agreement on G.

1.4 Contributions

In this work, we introduce the *abstract approximate agreement problem on a convexity space* $\mathcal{C}$ satisfying:

- agreement: Y is a free set, that is, $\langle Y \rangle = \operatorname{ex} Y$, where $\operatorname{ex} Y$ is the extreme points of Y.
- validity: $Y \subseteq \langle X \rangle$.

While our primary focus lies in the graphical version of approximate agreement, we believe the abstract problem is also interesting in itself. Indeed, it conveniently turns out that the problem coincides with various natural agreement problems: In graphs, the monophonic and geodesic approximate agreement on graphs problem given above boils down to solving approximate agreement on the chordless path or geodesic convexities of G. Moreover, lattice agreement on $\mathbb{L}$ is equivalent to solving approximate agreement on the *algebraic convexity space* of the semilattice (sets closed under $\oplus$). Our key results can be summarised as follows:

[1] A path is *chordless* (also known as *minimal*) if there are no edges between non-consecutive vertices.
[2] A graph is a *block graph* if every 2-connected component is a clique.

1. **Byzantine approximate agreement on chordal graphs.** We give algorithms for approximate agreement on trees and chordal graphs. The algorithms tolerate $f < n/(\omega+1)$ Byzantine faults and terminate in $O(\log N)$ *asynchronous* rounds, where ω is the clique number and N is the number of vertices in the value graph G. In trees, we achieve optimal resilience.
2. **Byzantine lattice agreement on cycle-free semilattices.** As another example, we give an asynchronous lattice agreement algorithm on cycle-free lattices that tolerates up to $f < n/(\omega + 1)$ Byzantine faults, where ω is the height of the semilattice. To our knowledge, this is the first algorithm that solves any variant of semilattice agreement under Byzantine faults.
3. **General impossibility results for asynchronous systems.** We give impossibility results for approximate agreement on arbitrary convex geometries parameterised by two combinatorial convexity invariants: the Carathéodory number c and the Helly number ω. As corollaries, we obtain resilience lower bounds for approximate agreement problems in asynchronous systems.
4. **Optimal synchronous algorithms for convex consensus.** We consider the *exact* variant of the abstract approximate agreement problem, where the agreement constraint is replaced by $|Y| = 1$. While the problem cannot be solved in asynchronous systems, we show that it can be solved on any convex geometry $\mathcal{C}$ in $\Theta(f)$ synchronous rounds if and only if $n > \omega f$ holds, where ω is the Helly number of $\mathcal{C}$. Moreover, the upper bound holds for *any* convexity space.

Our work can be seen as an extension of the Mendes–Herlihy approximate agreement and Vaidya–Garg multidimensional consensus frameworks [46, 54, 47] onto general convexity spaces. However, while these operate in continuous m-dimensional Euclidean spaces, our analysis relies on combinatorial theory of abstract convexity, where the input and output values have discrete, combinatorial structure. In particular, the discrete nature of the convexity space poses new challenges, as unlike in the continuous setting, non-trivial convex sets do not necessarily contain non-extreme points to choose from to facilitate convergence.

Multidimensional agreement problems in Euclidean spaces have applications ranging from, e.g., robot convergence tasks to distributed voting and convex optimisation [47]. Our work extends the scope of these techniques to discrete convexity spaces, which can be used to describe various natural combinatorial systems. Finally, unlike prior work, our algorithms do not assume that processors can perform computations or send messages involving arbitrary precision real values, as in the discrete case a single value can be encoded using $O(\log |V|)$ bits.

1.5 Related work

The seminal result of Fischer et al. [28] showed that *exact* consensus cannot be reached in asynchronous systems in the presence of crash faults. Dolev et al. [14] showed that it is however possible to reach *approximate agreement* in an asynchronous system even with arbitrary faulty behavior when the values reside on the continuous real line. Subsequently, the one-dimensional approximate agreement problem has been extensively studied [14, 26, 27, 1]. Fekete [27] showed that any algorithm reducing the distance of values from d to ε requires $\Omega(\log(\varepsilon/d))$ asynchronous rounds when $f \in \Theta(n)$; in the discrete setting this yields the bound $\Omega(\log N)$ for paths of length N. Recently, Mendes et al. [47] introduced the natural generalisation of *multidimensional* approximate agreement and showed that the m-dimensional problem is solvable in an asynchronous system with Byzantine faults if and only if $n > (m + 2)f$ holds for any given $m \geq 1$.

The *lattice agreement problem* was originally introduced in the context of wait-free algorithms in shared memory models [3, 2]. The problem has recently resurfaced in the context of asynchronous message-passing models with crash faults [23, 58]. These papers consider the problem when the validity condition is given as $x_i \leq y_i \leq \bigoplus\{x_j : j \in P\}$, i.e., the output of a processor must satisfy $x_i \leq y_i$ and the feasible area is determined also by the inputs of faulty processors. However, it is not difficult to see that under Byzantine faults, this validity condition is not reasonable, as the problem cannot be solved even with one faulty processor.

Another class of structured agreement problems in the wait-free asynchronous setting are *loop agreement* tasks [32], which generalise k-set agreement and approximate agreement (e.g., $(3, 2)$-set agreement and one-dimensional approximate agreement). In loop agreement, the set of inputs consists of three distinct vertices on a loop in a 2-dimensional simplicial complex and the outputs are vertices of the complex with certain constraints, whereas *rendezvous tasks* are a generalisation of loop agreement to higher dimensions [43]. These tasks are part of large body of work exploring the deep connection of asynchronous computability and combinatorial topology, which has successfully been used to characterise the *solvability* of various distributed tasks [31]. Gafni and Kuznetsov's P-reconciliation task [29] achieves geodesic approximate agreement on a graph of system configurations.

Finally, we note that distributed agreement tasks play a key role in many fault-tolerant clock synchronisation algorithms [57, 42, 41]. Byzantine-tolerant clock synchronisation can be solved using one-dimensional approximate agreement [57], whereas in the *self-stabilising* setting both exact digital clock synchronisation [41] and pulse synchronisation tasks reduce to consensus [42]. However, while the latter problem has been extensively studied [16, 13, 42], non-trivial lower bounds are still lacking [42]. Given that clock synchronisation closely relates to agreement on cyclic structures, investigating agreement tasks on different structures may yield insight into the complexity of fault-tolerant (approximate) clock synchronisation. Indeed, we show that agreement on graphs *without* long induced cycles is considerably easier than consensus.

2 Preliminaries

We start with some basic preliminaries needed to describe the main ideas and results of the paper.

2.1 Abstract convexity spaces

Let V be a finite set. The collection $\mathcal{C} \subseteq 2^V$ is a *convexity space on* V if (1) $\emptyset, V \in \mathcal{C}$ holds, and (2) $A, B \in \mathcal{C}$ implies that $A \cap B \in \mathcal{C}$. A set $K \in \mathcal{C}$ is said to be *convex*. For any $A \subseteq V$, the *convex hull* of A is the minimal convex set $\langle A \rangle \in \mathcal{C}$ such that $A \subseteq \langle A \rangle$. Thus, $\langle \cdot \rangle$ is a closure operator on V. For any $A \subseteq V$ and $a \in A$, a is called an *extreme point* of A if $a \notin \langle A \setminus a \rangle$. For a convex set $K \in \mathcal{C}$, we use $\operatorname{ex} K$ to denote the extreme points of K. The convexity space $\mathcal{C}$ is a *convex geometry* if every $K \in \mathcal{C}$ satisfies $K = \langle \operatorname{ex} K \rangle$. A convex set K is *free* if $K = \operatorname{ex} K$. Finally, a nonempty set $A \subseteq V$ is *irredundant* if $\partial A \neq \emptyset$ where $\partial A = \langle A \rangle \setminus \bigcup_{a \in A} \langle A \setminus a \rangle$. The following theorem characterises convex geometries:

▶ **Theorem 1** ([21]). *Let $\mathcal{C}$ be a convexity space on V. The following conditions are equivalent:*

1. *$\mathcal{C}$ is a convex geometry.*
2. *For every $K \in \mathcal{C}$, $K = \langle \operatorname{ex} K \rangle$ (Minkowski-Krein-Milman property).*
3. *For every $K \in \mathcal{C} \setminus \{V\}$, there exists an element $u \in V \setminus K$ such that $K \cup \{u\} \in \mathcal{C}$.*

2.1.1 Carathéodory and Helly numbers

The *Carathéodory number* of a convexity space $\mathcal{C}$ on V is the smallest integer c such that for any $U \subseteq V$ and any $u \in \langle U \rangle$, there is a set $S \subseteq U$ such that $|S| \leq c$ and $u \in \langle S \rangle$. The Carathéodory number of a convexity space equals the maximum size of an irredundant set in $\mathcal{C}$. A collection $\mathcal{C}$ of sets is k-intersecting if every $\mathcal{B} \subseteq \mathcal{C}$ with $|\mathcal{B}| \leq k$ has a nonempty intersection. The *Helly number* of a convexity space $\mathcal{C}$ is the smallest integer ω such that any finite ω-intersecting $\mathcal{A} \subseteq \mathcal{C}$ has a nonempty intersection. If $\mathcal{C}$ is a convex geometry, the Helly number equals the maximum cardinality of a free set in $\mathcal{C}$ [21].

2.1.2 Examples of convex geometries

In this work, we focus on the following convexity spaces:

- Let $G = (V, E)$ be a graph. A set $U \subseteq V$ is convex if all the vertices on all minimal, i.e., chordless, paths connecting any $u, v \in U$ are contained in U. The Helly number of this convexity space equals the size of the maximum clique in G [34, 19] and the Carathéodory number is at most two [19]. Moreover, free sets coincide with cliques. The convexity space is a convex geometry iff G is chordal [24]. Indeed, if G contains an induced cycle K of length at least four, then it is easy to check that K is convex, but has no extreme points.
- Let $\mathbb{L} = (V, \oplus)$ be a semilattice and $\mathcal{C}$ be the collection of sets closed under $\oplus$. The collection $\mathcal{C}$ is a convex geometry, where every $K \in \mathcal{C}$ corresponds to a subsemilattice $(K, \oplus)$ of $\mathbb{L}$. A set K is free if and only if it is a chain [51]. Thus, the Helly number of a semilattice equals its height. Moreover, the Carathéodory number equals the breadth of the semilattice.

2.2 Asynchronous rounds

When operating in the asynchronous model, we describe and analyse the algorithms in the *asynchronous round* model. In this model, each processor has a local round counter and labels all of its messages with a round number. Each correct node initialises its round counter to 0 at the start of the execution and increases its local round counter from t to $t+1$ only when it has received at least $n-f$ messages belonging to round t (since up to f faulty nodes may omit their messages). In each round $t \geq 0$, a non-faulty processor i

1. sends a value to each processor $j \in P$,
2. receives a value $M_{ij}(t)$ from each processor $j \in P$,
3. updates local state and proceeds to round $t+1$.

The received message $M_{ij}(t)$ may be empty, denoted by $\perp$, to indicate that no message arrived from processor j (e.g., due to a crash or a delay). We use the set

$$P_i(t) = \{j \in P : M_{ij}(t) \neq \perp\}$$

to denote the processors from which i received a nonempty message on round t.

Assuming $n > 3f$ and with the help of reliable broadcast, the witness technique [1, 47], and attaching round numbers to all messages, the Byzantine asynchronous round model can guarantee the following for each $i, j \in P \setminus F$:

1. $|P_i(t)| \geq n - f$,
2. $|P_i(t) \cap P_j(t)| \geq n - f$,
3. if $M_{ik}(t) = x \neq \perp$ for $k \in F$, then $M_{jk}(t) \in \{x, \perp\}$.

That is, (1) every correct processor receives at least $n-f$ nonempty values (out of which f may be from faulty processors), (2) any two correct processors receive at least $n-f$ common values (possibly f of which may be from faulty processors), and (3) if some correct processor receives a nonempty value x from a faulty processor, then all other correct processors receive the same value or no value. The Byzantine asynchronous round model can be simulated in the asynchronous model so that a non-faulty processor broadcasts $O(n \log n)$ additional bits per round [1, 47].

2.3 Graphs

Let $G=(V,E)$ be a finite undirected connected graph, where $V=V(G)$ denotes the set of vertices and $E=E(G)$ the set of edges. We assume all graphs are simple (no parallel edges) and loopless (no self-loops). For any $U \subseteq V$, we use $G[U]=(U,F)$, where $F=\{\{u,v\} \in E : u,v \in U\}$, to denote the subgraph of G induced by the vertices in U. An ℓ-length path $u \rightsquigarrow v$ from a vertex u to vertex v is a non-repeating sequence $(u=v_0,\dots,v_\ell=v)$ of vertices such that $\{v_i,v_{i+1}\} \in E$. An ℓ-cycle is an $(\ell-1)$-path from u to v with $\{u,v\} \in E$. A path $v_0,\dots,v_\ell$ is *chordless* (or minimal) if there does not exist any edge $\{v_i,v_j\} \in E$ for $j>i+1$.

For vertices $u,v \in V$, we denote the length of the shortest path between u and v as $d(u,v)$. The eccentricity of vertex v is $\mathrm{ecc}(v)=\max\{d(v,u) : u \in V(G)\}$. For a set $U \subseteq V$, we define its diameter $D(U)=\max\{d(u,v) : u,v \in U\}$. The diameter of graph G is denoted by $D(G)=D(V)$. The *radius* of a graph G is $R(G)=\min\{\mathrm{ecc}_G(v) : v \in V(G)\}$ and the *center* is $\mathrm{center}(G)=\{u \in V(G) : \mathrm{ecc}_G(u)=R(G)\} \subseteq V(G)$. For a connected set $U \subseteq V$, we use the short-hands $R(U)=R(G[U])$ and $\mathrm{center}(U)=\mathrm{center}(G[U])$.

A graph G is a *tree* if it contains no cycles and *chordal* if contains no ℓ-cycle with $\ell \geq 4$ as an induced subgraph. A graph is *Ptolemaic* if it is chordal and distance-hereditary (any connected induced subgraph preserves distances). The clique number $\omega(G)$ is the size of the largest clique in G. A vertex is *simplicial* in $U \subseteq V$ if its neighbourhood $\mathcal{N}(v) \cap U$ in U is a clique. A perfect elimination ordering $\preceq$ of $G=(V,E)$ is a total order on V such that any $u \in V$ is simplicial in $\{v : u \preceq v\}$. A graph has a perfect elimination ordering iff it is chordal.

2.3.1 Chordal graphs

Chordal graphs (also known as triangulated, rigid or decomposable graphs) form an important and well-studied class of graphs. From a structural point of view, they have many equivalent characterisations: they are graphs that have no induced cycles greater than three, graphs for which perfect elimination orderings exist, graphs in which every minimal vertex separator is a clique, and others [12, 52, 30, 53, 24].

Due to their ubiquitous nature and convenient structural properties, the algorithmic aspects of chordal graphs have received much attention in the past decades. For example, chordal graphs have applications in a variety of contexts including combinatorial and semi-definite optimisation [56] and probabilistic graphical models [40]. Indeed, many NP-hard problems, such as finding maximum cliques or optimal vertex colourings, often admit simple polynomial time solutions in chordal graphs [30]. In the distributed setting, it is possible to find good approximations to minimum vertex colourings and maximum independent sets in chordal graphs [37].

2.4 Lattices

A (join) semilattice is an idempotent commutative semigroup $\mathbb{L} = (V, \oplus)$, where V is a finite set and $\oplus : V \times V \to V$ is called the *join* operator. A semilattice has a natural partial order defined as $u \leq v \iff u \oplus v = v$, where $u \oplus v$ is the least upper bound (i.e., a join) of $\{u, v\}$ in the partial order. We write $u < v$ if $u \leq v$ and $u \neq v$. For any set $U = \{u_0, \ldots, u_\ell\} \subseteq V$, the least upper bound $\bigoplus U$ is known as the join of U. If $u_0 \leq \ldots \leq u_\ell$ holds, then U is said to be a chain from u_0 to u_ℓ. If $\{u, v\}$ is a chain, then u and v are said to be comparable. The height $\omega(\mathbb{L})$ of a semilattice is the maximum cardinality of any chain $U \subseteq V$. The breadth of a semilattice is the smallest integer b such that for any nonempty $U \subseteq V$ there is a subset $A \subseteq U$ of size at most b satisfying $\bigoplus A = \bigoplus U$.

3 Approximate agreement on abstract convexity spaces

3.1 Iterative algorithm on abstract convexity spaces

In this section, we describe a basic step for an approximate agreement algorithm in the Byzantine asynchronous round model in an abstract convexity space $\mathcal{C}$ with Helly number ω. The algorithm is a generalisation of the Mendes–Herlihy algorithm by Mendes et al. [47] onto abstract convexity spaces. It is not guaranteed, however, to converge on *all* discrete convexity spaces.

The algorithm proceeds iteratively. At the start of each asynchronous round t, each correct processor $i \in P \setminus F$ broadcasts its current value $x_i(t) \in V$. At the end of round t, processor i has received a value from at least $n - f$ processors $P_i(t) \subseteq P$. These values are used to compute a new value $x_i(t+1) = y_i(t)$. For brevity, we often omit t from our notation, e.g., use the short-hands such as $P_i = P_i(t)$.

3.1.1 Computing the safe area

For any subset of processors $J \subseteq P_i$, define

$$\mathcal{V}_i(J) = \{M_{ij}(t) \neq \bot : j \in J\}$$

to be the set of values processor i received from processors in J. Processor i locally computes

$$\mathcal{K}_i = \left\{ \langle \mathcal{V}_i(J) \rangle : J \in \binom{P_i}{|P_i| - f} \right\} \quad \text{and} \quad H_i = \bigcap \mathcal{K}_i,$$

where $\langle \cdot \rangle$ denotes the convex hull operator. Processor i then outputs the value

$$y_i = \begin{cases} \phi(H_i) & \text{if } H_i \neq \emptyset, \\ \bot & \text{otherwise,} \end{cases}$$

where $\phi \colon \mathcal{C} \to V$ is an output map, which will depend on the convexity space $\mathcal{C}$ we are working in, see Section 4 for an output map for chordal graphs. The Helly property guarantees that H_i and y_i remain in the closure $\langle X \rangle$ of the input values. For each $t \geq 0$, we define $X(t) = \{x_j(t) : j \in P \setminus F\} \subseteq V$ and $Y(t) = \{y_j(t) = \phi(H_j(t)) : j \in P \setminus F\}$.

▶ **Lemma 2.** *Suppose $\mathcal{C}$ is a convexity space on V with Helly number ω. If $n > \max\{(\omega + 1)f, 3f\}$ holds, then for each iteration $t \geq 0$ the above algorithm satisfies*
- $\emptyset \neq H_i(t) \subseteq \langle X(t) \rangle$ *for all* $i \in P \setminus F$,
- $\bigcap_{i \in P \setminus F} H_i(t) \neq \emptyset$.

3.2 On the elimination of extreme points

If we can in each iteration remove some extreme point of $\langle X \rangle$, where X is the set of input values, then the hull of output values $\langle Y \rangle$ shrinks. In an arbitrary convexity space, a convex set may not have any extreme points (consider, e.g., the chordless path convexity on a four cycle). However, in a convex geometry every nonempty convex set has an extreme point, as $\langle X \rangle = \langle \operatorname{ex} X \rangle$ by Theorem 1.

Moreover, finite convex geometries are in a sense "easy to peel" iteratively – at least from the global perspective. We say that a total order $\preceq$ on V is a *convex elimination order* of $\mathcal{C}$ if for any $u \in V$ the sets $A(u) = \{v \in V : u \preceq v\}$ and $A(u) \setminus u$ are convex. Theorem 1 implies that convex geometries admit such orderings and we assume that the values in V are labelled according to such an order $\preceq$. For a set $U \subseteq V$, we let $\max U$ and $\min U$ be the unique maximal and minimal elements, respectively, given by $\preceq$ such that for all $v \in U$ we have $\min U \preceq v$ and $v \preceq \max U$.

▶ Remark 3. For any $K \subseteq V$ it holds that $\min K \in \operatorname{ex} K$ and $\min K = \min\langle K \rangle$.

The next lemma shows that to guarantee progress by shrinking the set of output values, it suffices to always exclude, e.g., $\min X$ from the output (of course, this is not indefinitely possible).

▶ **Lemma 4.** *If $\min X \notin Y$ in a convex elimination order, then $\langle Y \rangle \subsetneq \langle X \rangle$.*

4 Approximate agreement on chordal graphs

We now show that monophonic approximate agreement on chordal graphs can be solved given that $n > (\omega + 1)f$ holds, where ω is the clique number of G. This also implies that geodesic approximate agreement is solvable on Ptolemaic graphs. Throughout we assume that $G = (V, E)$ is a connected chordal graph with at least two vertices and $\mathcal{C}$ is its chordless path convexity space. We recall that the Helly number of $\mathcal{C}$ coincides with the clique number $\omega = \omega(G)$ of G [34, 19].

4.1 Approximate agreement on trees

Suppose $G = (V, E)$ is a tree. As G is also chordal, it has a perfect elimination ordering $\preceq$. We assume that the vertices of V are labelled according to this ordering and define the output map

$$\phi(K) = \max \operatorname{center} K,$$

where $\operatorname{center} K \subseteq V$ is the center of the subgraph $G[K]$ induced by K. This rule roughly divides the diameter of the set of active values by two, which yields the following result.

▶ **Theorem 5.** *If $n > 3f$ and $G = (V, E)$ is a tree, then approximate agreement on G can be solved in $\log D(G) + 1$ asynchronous rounds, where $D(G)$ is the diameter of G.*

4.2 Fast monophonic approximate agreement on chordal graphs

We now present a fast monophonic approximate agreement algorithm on chordal graphs. To this end, we use the tree algorithm above on a suitable tree decomposition of the actual graph G.

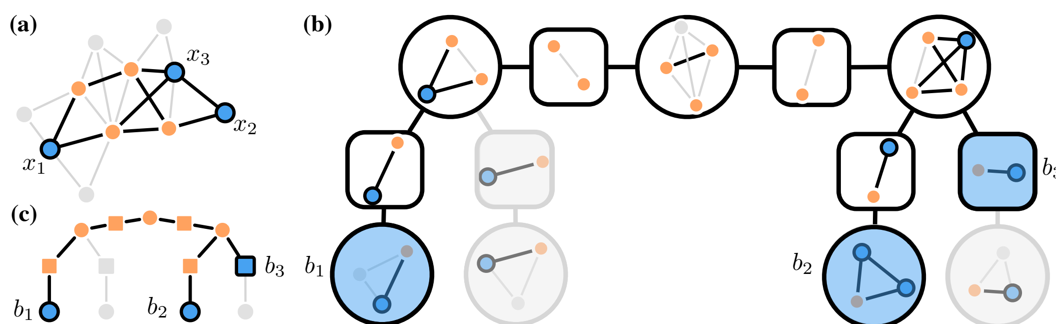


Figure 2 Approximate agreement on chordal graphs via clique trees. (a) The chordal value graph G and three input values $X = \{x_1, x_2, x_3\} \subseteq V(G)$. (b) An expanded clique tree (T, χ) of G. The bags $B = \{b_1, b_2, b_3\}$ satisfy $x_i \in \chi(b_i)$ and the bags are used as input for the approximate agreement algorithm on trees. The round bags are maximal cliques and the rectangular bags are the intersection of its neighbouring round bags, which are the minimal vertex separators in G. Note that a bag $b \in \langle B \rangle$ may contain vertices of G outside the convex hull $\langle X \rangle$ of the initial input values X (e.g., bag b_1 and the central bag). (c) The graph T on which we run the tree algorithm.

▶ **Definition 6.** *Let G be a graph, T a tree and $\chi \colon V(T) \to 2^{V(G)}$ be a mapping. We say that the pair (T, χ) is a* tree decomposition *of G if the following conditions are satisfied:*
- *for all $v \in V(G)$ there exists $b \in V(T)$ such that $v \in \chi(b)$,*
- *for all $e \in E(G)$ there exists $b \in V(T)$ such that $e \subseteq \chi(b)$,*
- *if $v \in \chi(a) \cap \chi(b)$, then $v \in \chi(c)$ for all $c \in V(T)$ residing on the unique path $a \rightsquigarrow b$.*

The tree decomposition is a clique tree *if each $b \in V(T)$ induces a maximal clique $\chi(b)$ in G.*

Chordal graphs can be characterised as those graphs having a clique tree [10, Proposition 12.3.11]. In fact, if G is chordal, the tree T can always be chosen as a spanning tree of G's clique graph, i.e., the graph whose nodes are the maximal cliques of G and whose edges join those cliques with nonempty intersection [7, Theorem 3.2]. Unlike non-chordal graphs in which the number of maximal cliques can be exponential, the number of maximal cliques is at most linear in chordal graphs:

▶ **Lemma 7** ([5])**.** *If G is a chordal graph, then it has a clique tree (T, χ) with $|V(T)| \in O(|V(G)|)$.*

For the purposes of our algorithm, we use a special kind of clique trees, which we call *expanded.* A clique tree (T, χ) is *expanded* if for each $\{a, b\} \in E(T)$ we have either $\chi(a) \subseteq \chi(b)$ or $\chi(b) \subseteq \chi(a)$; see Figure 2a–b for an example of an expanded clique tree.

▶ **Lemma 8.** *Every chordal graph G has an expanded clique tree T with $|V(T)| = O(|V(G)|)$.*

4.2.1 The algorithm

Let $G = (V, E)$ denote the chordal value graph, (T, χ) an expanded clique tree of G, and $\mathbf{A}$ the approximate agreement algorithm on trees given by Theorem 5. Given an input $x_i(0) \in V(G)$ on the graph G, processor $i \in P \setminus F$ starts by choosing any *bag* $b_i(0) \in V(T)$ such that the $x_i(0) \in b_i(0)$; see Figure 2. In iteration $t \geq 1$, every processor $i \in P \setminus F$ performs the following:

1. Broadcast $x_i(t)$ and $b_i(t)$ to all other processors.
2. Simulate one step of $\mathbf{A}$ on the $b(\cdot)$ values and set $b_i(t+1) = \mathbf{A}\left(b_{0,i}(t), \ldots, b_{n-1,i}(t)\right)$.
3. Compute the safe area H_i^G from the received values $x_{ij}(t)$.
4. Set $x_i(t+1)$ to an arbitrarily chosen element of $\chi(b_i(t+1)) \cap H_i^G$.

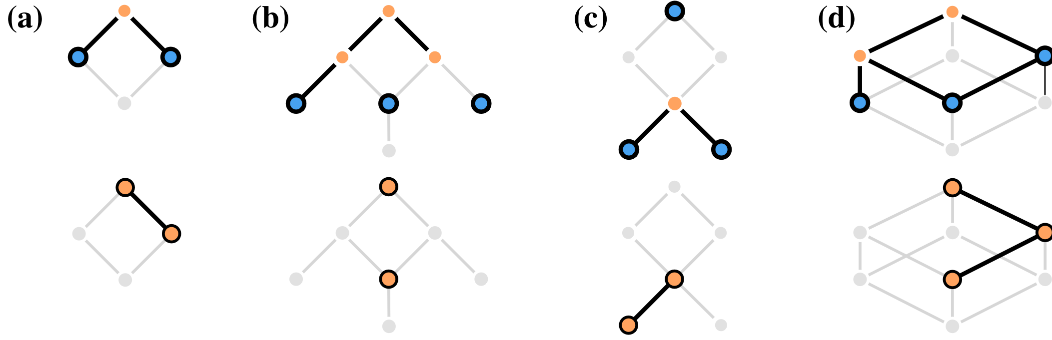


Figure 3 Examples of algebraic convex sets on semilattices. The figures show the Hasse diagrams of these semilattices. In the top row, the blue vertices are input values and the orange vertices are contained in the hull of the blue vertices. The bottom row shows feasible outputs for these cases (i.e. chains contained in the convex hull). The semilattices (a)–(b) are cycle-free, whereas (c) and (d) respectively contain an induced 4- and 6-cycle in the comparability graph.

Since the $b_i(\cdot)$ values are updated using the algorithm **A**, these values converge onto a single edge $\{a, b\} \in E(T)$ in the tree T. As $\chi(a) \cup \chi(b)$ is a clique due to the expandedness of T, the output values $x(\cdot)$ will have diameter at most one in G assuming that $x_i(t+1)$ is well-defined for each $i \in P_i \setminus F$, that is, $\chi(b_i(t+1)) \cap H_i^G \neq \emptyset$. Showing this is the main challenge of the correctness proof.

▶ **Theorem 9.** *Let $G = (V, E)$ be a chordal graph. If $n > (\omega(G) + 1)f$, then monophonic approximate agreement on G can be solved in $O(\log|V|)$ asynchronous rounds.*

Finally, we observe that the above implies that *geodesic* approximate agreement can be solved in Ptolemaic graphs, as geodesic and monophonic convexities are identical on these graphs [24].

▶ **Corollary 10.** *If $G = (V, E)$ is a connected Ptolemaic graph and $n > (\omega(G) + 1)f$, then geodesic approximate agreement on G is solvable in $O(\log |V|)$ asynchronous rounds.*

5 Byzantine lattice agreement on cycle-free semilattices

The abstract convex geometry framework can be easily applied to solve agreement problems on other combinatorial structures. As an example we consider asynchronous Byzantine lattice agreement on a special class of semilattices. Let $\mathbb{L} = (V, \oplus)$ be a semilattice and $\leq$ its natural partial order. The *comparability graph* of $\leq$ is the graph $G = (V, E)$ where $\{u, v\} \in E$ if $u \neq v$ and u and v are comparable. A partial order $\leq$ is *cycle-free* if the comparability graph is chordal [45]. Similarly, we say $\mathbb{L}$ is cycle-free if $\leq$ is cycle-free. See Figure 3 for examples of cycle-free and non-cycle-free semilattices.

▶ **Lemma 11.** *Let $\mathbb{L} = (V, \bigoplus)$ be a cycle-free semilattice. There exists an elimination order $\preceq$ on the algebraic convexity of $\mathbb{L}$ such that $A(u) = \{v : u \oplus v \in \{u, v\}, u \preceq v\}$ is a chain for any $u \in V$.*

We assume now that $\preceq$ is the ordering given by Lemma 11 and use the following output map

$$\phi(K) = \begin{cases} \bigoplus K & \text{if } K \neq \operatorname{ex} K \\ \max K & \text{otherwise.} \end{cases}$$

With this, the framework given in Section 3 and Lemma 4 yield the following result.

▶ **Theorem 12.** *Suppose $\mathbb{L} = (V, \oplus)$ is a cycle-free semilattice of height ω and $n > (\omega + 1)f$. Then Byzantine semilattice agreement on $\mathbb{L}$ can be solved in the asynchronous model.*

6 Resilience lower bounds for abstract convexity spaces

We obtain general lower bounds for asynchronous approximate agreement on abstract convexity spaces. We derived a general way of obtaining impossibility results using a partitioning argument and so-called blocking instances. This makes it possible to show how to obtain blocking instances for convex geometries from irredundant and free sets. Recall that the Carathéodory number c of $\mathcal{C}$ equals the maximum cardinality of an irredundant set in $\mathcal{C}$. This yields the following result, which can be seen as a generalisation of the Mendes et al. [47] lower bound technique from Euclidean spaces into arbitrary convexity spaces:

▶ **Theorem 13.** *Let $\mathcal{C}$ be a convexity space with Carathéodory number c and Helly number ω. Then:*

- *If $n \leq (c+1)f$, then there is no asynchronous abstract approximate agreement algorithm on $\mathcal{C}$ that on all inputs satisfies validity and agreement (i.e., the set of outputs is free).*
- *If $\mathcal{C}$ is a convex geometry and $n \leq (\omega + 1)f$, then there is no asynchronous abstract approximate agreement algorithm on $\mathcal{C}$ satisfying validity that outputs at most $\omega - 1$ distinct values.*

Combining the above result with classic results in combinatorial convexity theory gives lower bounds for specific problems. For any graph G with diameter at least two, the Carathéodory number is two [19] and clique is a free set. This implies the following result.

▶ **Corollary 14.** *The monophonic approximate agreement problem on any G with diameter at least two cannot be solved if $n \leq 3f$. There is no asynchronous algorithm that outputs a clique of size less than ω unless $n \leq (\omega + 1)f$.*

The case of Byzantine semilattice agreement is perhaps more interesting, as the breadth of a semilattice coincides with its Carathéodory number [35]. For any $b > 1$, there are semilattices with height and breadth equal to b: take the subsemilattice of a subset lattice over $[b]$ without $\emptyset$.

▶ **Corollary 15.** *Suppose $\mathbb{L}$ is a semilattice with breadth b. If $n \leq (b+1)f$, then there exists no asynchronous algorithm that solves Byzantine semilattice agreement on $\mathbb{L}$.*

7 Synchronous convex agreement

Finally, we give matching upper and lower bound results for exact convex consensus problem on abstract convexity spaces. It is easy to see that convex consensus is at least as hard as binary consensus, i.e., classic impossibility results for binary consensus [49, 28, 15] also hold for convex consensus. Hence, we consider the synchronous model of computation.

▶ **Theorem 16.** *Let $\mathcal{C}$ be an abstract convexity space on V with Helly number ω. If $n > \max\{3f, \omega f\}$, then convex consensus on $\mathcal{C}$ can be solved in $O(f)$ synchronous communication rounds using $O(nf^2)$ messages of size $O(n \cdot (\log n + \log |V|))$.*

It turns out that the higher resilience threshold of $n > \omega f$ for convex consensus is necessary already in the case of convex geometries.

▶ **Theorem 17.** *Let $\mathcal{C}$ be a convex geometry with a Helly number ω. If $n \leq \omega f$ holds, then convex consensus on $\mathcal{C}$ cannot be solved in the synchronous message-passing model.*

8 Conclusions

Many structured agreement tasks correspond to exact or approximate agreement problems on (possibly discrete) convexity spaces. Using the theory of abstract convexity, we have obtained Byzantine-tolerant algorithms for a large class of agreement problems on discrete combinatorial structures. In the synchronous model, exact convex consensus for any convexity space can be solved in an optimally resilient manner with asymptotically optimal round complexity. However, in the asynchronous setting, several interesting open problems remain.

1. It seems difficult to come up with a general rule for the output map $\phi : \mathcal{C} \to V$ in a way that guarantees that the convex hull of active values shrinks. Nevertheless, we have seen that on chordal graphs and cycle-free semilattices we can solve approximate agreement efficiently. In both cases, the underlying convexity space is a convex geometry. Given that the literature is abound with convex geometries associated with combinatorial structures [34, 24, 25, 19, 50, 48, 17, 18, 21, 51], it is natural to ask whether the abstract approximate agreement problem can be solved for other convex geometries as well.
2. It is unclear whether the abstract approximate agreement problem can be solved on general convexity spaces. For example, the asynchronous algorithms for approximate agreement on graphs presented here fail for non-chordal graphs: already the simplest example of a non-chordal graph, the four cycle, is difficult to handle. Indeed, the monophonic convexity of a four cycle is *not* a convex geometry: a convex set may not necessarily have any extreme points, and thus, greedily excluding extreme points does not seem to work. Are there resilient asynchronous algorithms that solve the problem for non-chordal graphs?
3. We obtained resilience lower bounds in terms of the Carathéodory and the Helly numbers. However, our positive results for the asynchronous model hold in cases where the Carathéodory number is at most two. Interestingly, in the continuous setting of multidimensional approximate agreement [47], tight resilience bounds exist, as the Carathéodory and Helly numbers coincide in the usual Euclidean convexity space on $\mathbb{R}^m$. Is there a discrete convexity space with a higher Carathéodory number in which approximate agreement can be solved?

References

1 Ittai Abraham, Yonatan Amit, and Danny Dolev. Optimal Resilience Asynchronous Approximate Agreement. In *Proc. International Conference on Principles of Distributed Systems (OPODIS 2015)*, pages 229–239, 2005. doi:10.1007/11516798_17.

2 Hagit Attiya and Arie Fouren. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM Journal on Computing*, 31(2):642–664, 2001.

3 Hagit Attiya, Maurice Herlihy, and Ophir Rachman. Atomic snapshots using lattice agreement. *Distributed Computing*, 8(3):121–132, 1995.

4 Rommel M Barbosa, Erika M. M. Coelho, Mitre C. Dourado, Dieter Rautenbach, and Jayme L. Szwarcfiter. On the Carathéodory number for the convexity of paths of order three. *SIAM Journal on Discrete Mathematics*, 26(3):929–939, 2012.

5 Anne Berry and Romain Pogorelcnik. A simple algorithm to generate the minimal separators and the maximal cliques of a chordal graph. *Information Processing Letters*, 111:508–511, 2011.

6 Martin Biely, Peter Robinson, and Ulrich Schmid. Easy impossibility proofs for k-set agreement in message passing systems. In *Proc. International Conference on Principles of Distributed Systems (OPODIS 2011)*, pages 299–312. Springer, 2011.

7 Jean R. S. Blair and Barry Peyton. An Introduction to Chordal Graphs and Clique Trees. In Alan George, John R. Gilbert, and Joseph W. H. Liu, editors, *Graph Theory and Sparse Matrix Computation*, pages 1–29. Springer, Heidelberg, 1993.

8 Soma Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132–158, 1993.

9 Roberto De Prisco, Dahlia Malkhi, and Michael Reiter. On k-set consensus problems in asynchronous systems. *IEEE Transactions on Parallel and Distributed Systems*, 12(1):7–21, 2001.

10 Reinhard Diestel. *Graph Theory.* Springer, Heidelberg, 4th edition, 2010.

11 Brenda L. Dietrich. Matroids and antimatroids – a survey. *Discrete Mathematics*, 78(3):223–237, 1989.

12 Gabriel Andrew Dirac. On rigid circuit graphs. In *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg*, volume 25, pages 71–76. Springer, 1961.

13 Danny Dolev, Matthias Függer, Christoph Lenzen, and Ulrich Schmid. Fault-tolerant algorithms for tick-generation in asynchronous logic. *Journal of the ACM*, 61(5):30:1–30:74, 2014. doi:10.1145/2560561.

14 Danny Dolev, Nancy A. Lynch, Shlomit S. Pinter, Eugene W. Stark, and William E. Weihl. Reaching Approximate Agreement in the Presence of Faults. *Journal of the ACM*, 33(3):499–516, May 1986. doi:10.1145/5925.5931.

15 Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for Byzantine agreement. *Journal of the ACM*, 32(1):191–204, 1985. doi:10.1145/2455.214112.

16 Shlomi Dolev and Jennifer L. Welch. Self-stabilizing clock synchronization in the presence of Byzantine faults. *Journal of the ACM*, 51(5):780–799, 2004. doi:10.1145/1017460.1017463.

17 Mitre C. Dourado, Dieter Rautenbach, Vinícius Fernandes dos Santos, Philipp M. Schäfer, and Jayme L. Szwarcfiter. On the Carathéodory number of interval and graph convexities. *Theoretical Computer Science*, 510:127–135, 2013. doi:10.1016/j.tcs.2013.09.004.

18 Pierre Duchet. Convexity in combinatorial structures. In *Proc. Winter School on Abstract Analysis*, pages 261–293. Circolo Matematico di Palermo, 1987.

19 Pierre Duchet. Convex sets in graphs, II. Minimal path convexity. *Journal of Combinatorial Theory, Series B*, 44(3):307–316, 1988. doi:10.1016/0095-8956(88)90039-1.

20 Jürgen Eckhoff. Helly, Radon, and Carathéodory type theorems. In *Handbook of Convex Geometry, Part A*, pages 389–448. Elsevier Science Publishers B.V, 1993.

21 Paul H. Edelman and Robert E. Jamison. The theory of convex geometries. *Geometriae Dedicata*, 19(3):247–270, 1985. doi:10.1007/BF00149365.

22 Paul H. Edelman and Michael E. Saks. Combinatorial representation and convex dimension of convex geometries. *Order*, 5(1):23–32, 1988. doi:10.1007/BF00143895.

23 Jose M. Faleiro, Sriram Rajamani, Kaushik Rajan, G. Ramalingam, and Kapil Vaswani. Generalized lattice agreement. In *Proc. ACM Symposium on Principles of Distributed Computing (PODC 2012)*, pages 125–134. ACM, 2012. doi:10.1145/2332432.2332458.

24 M. Farber and R. E. Jamison. Convexity in graphs and hypergraphs. *SIAM Journal on Algebraic Discrete Methods*, 7(3):433–444, 1986.

25 Martin Farber and Robert E. Jamison. On local convexity in graphs. *Discrete Mathematics*, 66(3):231–247, 1987. doi:10.1016/0012-365X(87)90099-9.

26 Alan David Fekete. Asymptotically optimal algorithms for approximate agreement. *Distributed Computing*, 4(1):9–29, 1990.

27 Alan David Fekete. Asynchronous approximate agreement. *Information and Computation*, 115(1):95–124, 1994.

28 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985. doi:10.1145/3149.214121.

29 Eli Gafni and Petr Kuznetsov. N-consensus is the second strongest object for $N+1$ processes. In *Proc. International Conference on Principles of Distributed Systems (OPODIS 2007)*, pages 260–273. Springer, Heidelberg, 2007.

30 Fănică Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SIAM Journal on Computing*, 1(2):180–187, 1972.

31 Maurice Herlihy, Dmitry Kozlov, and Sergio Rajsbaum. *Distributed Computing Through Combinatorial Topology*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013.

32 Maurice Herlihy and Sergio Rajsbaum. A classification of wait-free loop agreement tasks. *Theoretical Computer Science*, 291(1):55–77, 2003.

33 Maurice Herlihy, Sergio Rajsbaum, and Mark R. Tuttle. Unifying synchronous and asynchronous message-passing models. In *Proc. ACM Symposium on Principles of Distributed Computing (PODC 1998)*, pages 133–142. ACM, 1998.

34 Robert E. Jamison and Richard Nowakowski. A Helly theorem for convexity in graphs. *Discrete Mathematics*, 51(1):35–39, 1984. doi:10.1016/0012-365X(84)90021-9.

35 Robert E Jamison-Waldner. A perspective on abstract convexity: classifying alignments by varieties. *Convexity and Related Combinatorial Geometry, New York*, 1982.

36 David Kay and Eugene W Womble. Axiomatic convexity theory and relationships between the Carathéodory, Helly, and Radon numbers. *Pacific Journal of Mathematics*, 38(2):471–485, 1971.

37 Christian Konrad and Viktor Zamaraev. Brief Announcement: Distributed Minimum Vertex Coloring and Maximum Independent Set in Chordal Graphs. In *Proc. ACM Symposium on Principles of Distributed Computing (PODC 2018)*, pages 159–161, New York, NY, USA, 2018. ACM. doi:10.1145/3212734.3212787.

38 Bernhard Korte and László Lovász. Greedoids – A Structural Framework for the Greedy Algorithm. In *Progress in Combinatorial Optimization*, pages 221–243. Academic Press, 1984. doi:10.1016/B978-0-12-566780-7.50019-2.

39 Bernhard Korte, László Lovász, and Rainer Schrader. *Greedoids*, volume 4. Springer Science & Business Media, 2012.

40 Steffen L. Lauritzen. *Graphical models*, volume 17. Clarendon Press, 1996.

41 Christoph Lenzen and Joel Rybicki. Near-optimal self-stabilising counting and firing squads. *Distributed Computing*, 2018. doi:10.1007/s00446-018-0342-6.

42 Christoph Lenzen and Joel Rybicki. Self-Stabilising Byzantine Clock Synchronisation is Almost as Easy as Consensus. *Journal of the ACM*, 2019. To appear.

43 Xingwu Liu, Zhiwei Xu, and Jianzhong Pan. Classifying rendezvous tasks of arbitrary dimension. *Theoretical Computer Science*, 410(21):2162–2173, 2009. doi:10.1016/j.tcs.2009.01.033.

44 László Lovăsz and Michael Saks. Communication complexity and combinatorial lattice theory. *Journal of Computer and System Sciences*, 47(2):322–349, 1993.

45 Tze-Heng Ma and Jeremy P. Spinrad. Cycle-free partial orders and chordal comparability graphs. *Order*, 8(1):49–61, March 1991. doi:10.1007/BF00385814.

46 Hammurabi Mendes and Maurice Herlihy. Multidimensional approximate agreement in Byzantine asynchronous systems. *Proc. Annual ACM symposium on Symposium on Theory of Computing (STOC 2013)*, page 391, 2013. doi:10.1145/2488608.2488657.

47 Hammurabi Mendes, Maurice Herlihy, Nitin Vaidya, and Vijay K. Garg. Multidimensional agreement in Byzantine systems. *Distributed Computing*, 28:423–441, 2015.

48 Morten H. Nielsen and Ortrud R. Oellermann. Steiner trees and convex geometries. *SIAM Journal on Discrete Mathematics*, 23(2):680–693, 2009.

49 Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980. doi:10.1145/322186.322188.

50 Ignacio M. Pelayo. *Geodesic Convexity in Graphs*. Springer, 2013. doi:10.1007/978-1-4614-8699-2.

51 Paul Poncet. Convexities on ordered structures have their Krein–Milman theorem. *Journal of Convex Analysis*, 21(1):89–120, 2014.

52 Donald J. Rose. Triangulated graphs and the elimination process. *Journal of Mathematical Analysis and Applications*, 32(3):597–609, 1970.

53 Donald J. Rose, R. Endre Tarjan, and George S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on computing*, 5(2):266–283, 1976.

54 Nitin H. Vaidya and Vijay K. Garg. Byzantine vector consensus in complete graphs. In *Proc. ACM Symposium on Principles of Distributed Computing (PODC 2013)*, pages 65–73, 2013.

55 Marcel LJ van De Vel. *Theory of convex structures*, volume 50. Elsevier, 1993.

56 Lieven Vandenberghe and Martin S Andersen. Chordal graphs and semidefinite optimization. *Foundations and Trends in Optimization*, 1(4):241–433, 2015.

57 Jennifer Lundelius Welch and Nancy Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation*, 77(1):1–36, 1988.

58 Xiong Zheng, Changyong Hu, and Vijay K. Garg. Lattice Agreement in Message Passing Systems. In *Proc. International Symposium on Distributed Computing (DISC 2018)*, volume 121 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 41:1–41:17, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.DISC.2018.41.

Small Cuts and Connectivity Certificates: A Fault Tolerant Approach

Merav Parter
Weizmann Institute, Rehovot, Israel
merav.parter@weizmann.ac.il

Abstract

We revisit classical connectivity problems in the CONGEST model of distributed computing. By using techniques from fault tolerant network design, we show improved constructions, some of which are even "local" (i.e., with $\widetilde{O}(1)$ rounds) for problems that are closely related to hard global problems (i.e., with a lower bound of $\Omega(Diam + \sqrt{n})$ rounds).

Distributed Minimum Cut: Nanongkai and Su presented a randomized algorithm for computing a $(1+\epsilon)$-approximation of the minimum cut using $\widetilde{O}(D + \sqrt{n})$ rounds where D is the diameter of the graph. For a sufficiently large minimum cut $\lambda = \Omega(\sqrt{n})$, this is tight due to Das Sarma et al. [FOCS '11], Ghaffari and Kuhn [DISC '13].

- **Small Cuts:** A special setting that remains open is where the graph connectivity λ is small (i.e., constant). The only lower bound for this case is $\Omega(D)$, with a matching bound known only for $\lambda \leq 2$ due to Pritchard and Thurimella [TALG '11]. Recently, Daga, Henzinger, Nanongkai and Saranurak [STOC '19] raised the open problem of computing the minimum cut in $\mathsf{poly}(D)$ rounds for any $\lambda = O(1)$. In this paper, we resolve this problem by presenting a surprisingly simple algorithm, that takes a completely different approach than the existing algorithms. Our algorithm has also the benefit that it computes *all* minimum cuts in the graph, and naturally extends to *vertex* cuts as well. At the heart of the algorithm is a graph sampling approach usually used in the context of fault tolerant (FT) design.
- **Deterministic Algorithms:** While the existing distributed minimum cut algorithms are randomized, our algorithm can be made deterministic within the same round complexity. To obtain this, we introduce a novel definition of universal sets along with their efficient computation. This allows us to derandomize the FT graph sampling technique, which might be of independent interest.
- **Computation of all Edge Connectivities:** We also consider the more general task of computing the edge connectivity of all the edges in the graph. In the output format, it is required that the endpoints u, v of every edge (u, v) learn the cardinality of the u-v cut in the graph. We provide the first sublinear algorithm for this problem for the case of *constant* connectivity values. Specifically, by using the recent notion of low-congestion cycle cover, combined with the sampling technique, we compute all edge connectivities in $\mathsf{poly}(D) \cdot 2^{O(\sqrt{\log n \log\log n})}$ rounds.

Sparse Certificates: For an n-vertex graph G and an integer λ, a λ-sparse certificate H is a subgraph $H \subseteq G$ with $O(\lambda n)$ edges which is λ-connected iff G is λ-connected. For D-diameter graphs, constructions of sparse certificates for $\lambda \in \{2, 3\}$ have been provided by Thurimella [J. Alg. '97] and Dori [PODC '18] respectively using $\widetilde{O}(D)$ number of rounds. The problem of devising such certificates with $o(D + \sqrt{n})$ rounds was left open by Dori [PODC '18] for any $\lambda \geq 4$. Using connections to fault tolerant spanners, we considerably improve the round complexity for *any* $\lambda \in [1, n]$ and $\epsilon \in (0, 1)$, by showing a construction of $(1-\epsilon)\lambda$-sparse certificates with $O(\lambda n)$ edges using only $O(1/\epsilon^2 \cdot \log^{2+o(1)} n)$ rounds.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms; Theory of computation → Distributed algorithms

Keywords and phrases Connectivity, Minimum Cut, Spanners

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.30

Funding *Merav Parter*: Support in part by an ISF grant no. 2084/18.

© Merav Parter;
licensed under Creative Commons License CC-BY

Acknowledgements I am thankful to DISC '19 reviewers for valuable input, and to Michal Dory for preliminary discussions on these problems. I am also grateful to Oded Goldriech for sparking my curiosity on the derandomization of the fault tolerant sampling approach. Finally, special thanks to Moni Naor and Karthik C. S. for useful discussions on universal sets.

1 Introduction

The connectivity of a graph is one of the most fundamental concept in graph theory and network reliability. In this paper, we revisit some classical connectivity problems in the CONGEST model of distributed computing via the lens of *fault tolerant* network design. We mainly focus on two problems: exact computation of small[1] edge (or vertex) cuts; and the computation of sparse connectivity certificates. Both of these problems have been studied thoroughly in the literature, and surprisingly still admit critically missing pieces. By using techniques from fault tolerant network design we considerably improve the state-of-the-art, as well as provide the first *deterministic* distributed algorithms for these problems.

1.1 Small Cuts

In the distributed minimum cut problem, given a graph G with edge connectivity λ, the goal is to identify at least one minimum cut, that is a collection of λ edges whose removal disconnect the graph. In the output format, each vertex should learn at least one possible minimum (edge) cut. We start by providing a brief history for the problem[2].

A Brief History. *(I) Upper Bounds:* The first non-trivial distributed algorithm for the minimum cut problem was given by Ghaffari and Kuhn [10]. They presented a randomized algorithm for computing a $(2+\epsilon)$ approximation of minimum cut using $\widetilde{O}_\epsilon(D+\sqrt{n})$ rounds, with high probability. Shortly after, Nanongkai and Su [16] improved the approximation ratio to $(1+\epsilon)$ with roughly the same round complexity. Recently, Daga et. al [5] provided an exact algorithm with sublinear round complexity which improves up on the state of the art in the regime of large cuts (i.e., for $\lambda = n^{\Omega(1)}$).

(II) Lower Bounds: In their seminal paper, Das-Sarma et al. [23] presented a lower bound of $\widetilde{\Omega}(D+\sqrt{n})$ rounds for the computation of an α-approximation of a weighted minimum cut which holds even for graphs with diameter $D = O(\log n)$. This lower bound applies only for weighted graphs with large weighted minimum cut of size $\Omega(\sqrt{n})$. Ghaffari and Kuhn [11] extended this lower bounds in two ways. First, they considered a weaker setting for weighted minimum cuts where the edge weights correspond to capacities, and thus nodes can exchange $O(w \log n)$ bits over edges of weight w in each round. They showed that even in this weaker model, the α-approximation of minimum cut in λ-edge connected graphs with $\lambda = \Theta(\sqrt{n})$ and diameter $O(\log n)$ requires $\widetilde{\Omega}(\sqrt{n/(\alpha \cdot \lambda)})$ rounds[3]. Observe that since in this construction $\lambda = \Theta(\sqrt{n})$, this lower bound can also be stated as $\Omega(\sqrt{\lambda})$ rather than $\Omega(D+\sqrt{n})$. In their second extension, Ghaffari and Kuhn attempted to capture also unweighted simple graphs. Here, they showed a lower bound of $\widetilde{\Omega}(D+\sqrt{n/(\alpha \cdot \lambda)})$ rounds,

[1] By small we mean of constant size.

[2] Although historically, the lower bound by Das-Sarma et al. [23] appeared before the upper bound algorithms, we reverse the order of presentation here.

[3] In the conference version of [10], a lower bound of $\widetilde{\Omega}(D+\sqrt{n})$ was mistakenly claimed for *any* $\lambda \geq 1$ and graphs with diameter $D = O(\log n)$. This was later on fixed in a modified arXiv version [11] and in Ghaffari's thesis [9].

for any $\lambda \geq 1$ but only for graphs with diameter $D = 1/\lambda \cdot \sqrt{n/(\alpha \cdot \lambda)}$. Again[4], with such a larger diameter, one can alternatively state this lower bound as $\Omega(\lambda \cdot D)$, rather than $\widetilde{\Omega}(D + \sqrt{n})$.

Computation of Small Cuts. The conclusion from the above discussion is that we still do not have matching bounds for the distributed minimum cut problem in cases where either (i) the value of the weighted minimum cut is $o(\sqrt{n})$, or (ii) the unweighted diameter is $o(\sqrt{n})$.

As most real-world networks admit small cuts [26], we are in particular intrigued by the complexity of computing the cuts in unweighted graphs with constant connectivity. Pritchard and Thurimella [21] showed an $O(D)$-round randomized algorithm for cut values up to 2. The problem of devising an $\mathsf{poly}(D)$ round algorithm for any constant $\lambda = O(1)$ was recently raised by Daga et al. [5]:

> *"A special case that deserves attention is when the graph connectivity is small. For example, is there an algorithm that can check whether an unweighted network has connectivity at most k in $\mathsf{poly}(k, D, \log n)$? ... Bounds in these forms are currently known only for $k \leq 2$."*

We answer this question in the affirmative by presenting a $\mathsf{poly}(D)$-round algorithm for any constant connectivity $\lambda = O(1)$. This algorithm in fact computes all possible minimum cuts in G, in the sense that for any min-cut set E' there is at least one vertex in the graph that knows E'. Turning to *vertex* cuts, [21] showed an $O(D + \Delta/\log n)$ round algorithm for computing the cut vertices. No exact algorithm is known for the case where the vertex connectivity is at least two. Our algorithm can be easily adapted to compute deterministically the (exact) vertex cuts in $\mathsf{poly}(D \cdot \Delta)$ rounds where Δ is the maximum degree.

1.2 Sparse Connectivity Certificates

For a given unweighted n-vertex D-diameter graph $G = (V, E)$ and integer $\lambda \geq 1$, a *connectivity certificate* is a subgraph $H \subseteq G$ satisfying that it is λ-edge (or vertex) connected iff G is λ-edge (or vertex) connected. The certificate is said to be *sparse* if H has $O(\lambda n)$ edges. Sparse certificates were introduced by Nagamochi and Ibaraki [15]. Thurimella [24] gave the first distributed construction of λ-sparse certificates using $O(\lambda \cdot (D + \sqrt{n}))$ rounds in the CONGEST model. For $\lambda = 2$, Censor-Hillel and Dory [3] showed[5] the randomized construction of a certificate with $O(n)$ edges using $O(D)$ rounds. In [7], Dory considered the case of $\lambda = 3$, and showed the construction of a certificate with $O(n \log n)$ edges and $O(D \log^3 n)$ rounds. These algorithms are randomized and are based on the cycle space sampling technique of Pritchard and Thurimella [21]. The problem of designing sparse certificates for any $\lambda \geq 4$ using $\widetilde{O}(D)$ rounds was left open therein [7].

In this paper, we provide an easy solution for this problem which takes only $\widetilde{O}(\lambda)$ rounds for any λ. This is based on the observation that fault tolerant spanners are in fact sparse connectivity certificates. As a result we get that the problem of designing sparse certificates is *local* rather than global (i.e., does not depend on the graph diameter). In the Our Results section we also improve the round complexity into $\widetilde{O}(1)$ (i.e., independent on λ) by loosing a small factor in the approximation.

---

[4] Also here the conference version [10] mistakenly claimed that the lower bound works even for graphs with diameter $D = O(\log n)$, and this was fixed in [11, 9].

[5] [3] studied the problem of the minimum k-edge-connected spanning subgraph (k-EECS), which for unweighted graphs implies the computation of connectivity certificates with a small number of edges.

1.3 Our Results

Distributed Computation of Small Minimum Cuts. We consider an unweighted D-diameter graph $G = (V, E)$ with edge connectivity $\lambda = O(1)$. We show a $\mathsf{poly}(D)$-round randomized algorithm to compute the minimum cut whose high level description can be stated in just few lines: Fix a vertex s and apply $\mathsf{poly}(D)$ iterations, where in iteration i we do as follows. (i) Sample a subgraph $G_i \subseteq G$ by adding each edge e into G_i independently with some fixed probability p. (ii) Compute a truncated BFS tree rooted at s up to depth $O(\lambda D)$ in G_i, and (iii) let each vertex t collect its s-t path in this tree (if such exists). Finally, after applying this procedure for $\mathsf{poly}(D)$ iterations, each vertex t computes locally the s-t cut on the subgraph that it has collected. The argument shows that every vertex t that is separated from s by some minimum cut E', can compute this set of edges w.h.p.

▶ **Theorem 1.** *Let G be an $\lambda = O(1)$ connected D-diam graph and max degree Δ. There exists a randomized minimum cut algorithm that runs in $\mathsf{poly}(D)$ rounds. In addition, with a small modification it computes the minimum vertex cut in $\mathsf{poly}(D \cdot \Delta)$ rounds.*

The algorithm is in fact stronger. Every vertex t also learns a collection $(\lambda - 1)$ edge disjoint paths from s (i.e., an integral flow from s). In addition, we do not compute only one minimum cut but rather for each minimum cut in G, there is at least one vertex that learns it.

Deterministic Computation of Small Cuts. So-far, the distributed minimum cut computation was inherently randomized. The randomized component of the algorithm of Thm. 1 is in the initial graph sampling in each iteration. To derandomize it, we introduce a new variant of *universal-sets*. We use this notion to explicitly compute, in polynomial time, a collection of $\mathsf{poly}(D)$ subgraphs $G_1, \ldots, G_k$ that have the same key properties as those obtained by the sampling approach. The polynomial computation is done *locally* at each vertex and thus does not effect the round complexity.

▶ **Theorem 2.** *One can compute small cuts deterministically in $\mathsf{poly}(D)$ rounds.*

This derandomization technique can be used to derandomize all other algorithms that are based on the fault tolerant (FT) sampling technique (e.g., [6],[27]), and it is therefore of interest also for centralized algorithms. Independently of our work, Alon, Chechik and Cohen [2] also studied the derandomization of algorithms that are based on the FT-sampling approach, their solution is different than ours.

For a summary on the computation of small cuts, see Table 1.

Table 1 State of the art results for exact distributed computation of small cuts.

	Min-Cut Value λ	#Rounds	Type
Pritchard & Thurimella [21]	2 Edges	$O(D)$	Rand.
Pritchard & Thurimella [21]	1 Vertex	$O(D + \Delta)$	Rand.
This Work	$O(1)$ Edges	$\mathsf{poly}(D)$	Det.
This Work	$O(1)$ Vertices	$\mathsf{poly}(D \cdot \Delta)$	Det.

Computation of Edge-Connectivities. We then turn to consider the more general task of computing the edge connectivity of all graph edges, up to some constant bound $\lambda = O(1)$. For an edge $e = (u, v)$, the *edge connectivity* of e is the size of the u-v minimum (edge) cut in G. In the output format for each edge $e = (u, v)$, its endpoints are required to learn the edge

connectivity of e. Exact computation of all edge connectivites has been previously known only $\lambda \leq 2$ due to Pritchard and Thurimella [21]. They gave randomized algorithms for the case of $\lambda = 1, 2$ with round complexities of $O(D)$ and $O(D + \sqrt{n}\log^* n)$, respectively.

In this paper, we again take a completely different approach and show a *deterministic* algorithm with $\mathsf{poly}(D) \cdot 2^{\sqrt{\log n \log\log n}}$ rounds for computing all edge connectivities up to constant value of $\lambda = O(1)$. Our algorithm is based on two tools: (1) *low-congestion cycle cover* [17] and their distributed computation [18] ; and (2) the derandomization of the FT-sampling approach.

▶ **Theorem 3.** *For every D-diameter n-vertex graph $G = (V, E)$, w.h.p., the edge connectivity of all graphs edges up to $\lambda = O(1)$ can be computed in $\mathsf{poly}(D) \cdot 2^{O(\sqrt{\log n})}$ rounds. This algorithm can also be derandomized using $\mathsf{poly}(D) \cdot 2^{O(\sqrt{\log n \log\log n})}$ rounds.*

Sparse Connectivity Certificates. In the second part of the paper we consider the related problem of computing connectivity certificates. We first show that by a direct application of fault tolerant spanners, one can compute a λ-edge connectivity certificate with $O(\lambda n)$ edges using $\widetilde{O}(\lambda)$ rounds. This considerably improves and extends up on the previous constructions with $O(D)$ rounds that were limited *only* for $\lambda \in \{2, 3\}$.

▶ **Lemma 4.** *For every $\lambda \in \mathbb{N}_{\geq 1}$, there is a randomized algorithm that computes a λ connectivity certificate with $O(\lambda n)$ edges in $O(\lambda \log^{1+o(1)} n)$ rounds, with high probability.*

By plugging in the recent *deterministic* spanner construction of [12], one can compute λ-edge connectivity certificate deterministically with $\widetilde{O}(\lambda \cdot n)$ edges and $\lambda \cdot 2^{O(\sqrt{\log n})}$ rounds[6]. This answers the open problem raised by Dory [7] concerning the existence of efficient deterministic constructions of connectivity certificates.

To avoid the dependency in λ in the round complexity, we use the well known Karger's edge-sampling technique, and show:

▶ **Lemma 5.** *For every λ-connected graph and $\epsilon \in (0, 1)$, there is a randomized distributed algorithm that computes a $(1 - \epsilon)\lambda$ connectivity certificate with $O(\lambda n)$ edges in $O(1/\epsilon^2 \cdot \log^{2+o(1)} n)$ rounds, with high probability.*

Table 2 summarizes the state of the art. Note that if one uses fault tolerant spanners resilient for *vertex* faults, we get $\widetilde{O}(\lambda)$-round algorithm for computing λ-vertex-certificates[7] with $O(\lambda^2 n)$ edges. For clarity of presentation, we mainly focus on the edge-connectivity certificates, but our results naturally extend to the vertex case as well.

Table 2 State of the art result for distributed computation of sparse connectivity certificates.

	Edge Connectivity λ	**Certificate Size**	**#Rounds**	**Type**
Thurimella [24]	Any	$\lambda \cdot n$	$\widetilde{O}(\lambda \cdot (D + \sqrt{n}))$	Det.
Pritchard & Thurimella [21]	2	$2n$	$O(D)$	Rand.
Dory [7]	3	$O(n \log n)$	$O(D \cdot \log^3 n)$	Rand.
This Work	Any	$O(\lambda n)$	$O(\lambda \cdot \log^{1+o(1)} n)$	Rand.
This Work	Approx. $(1-\epsilon)$	$O(\lambda \cdot n)$	$O(\log^{2+o(1)} n)$	Rand.
This Work	Any	$\widetilde{O}(\lambda \cdot n)$	$\lambda \cdot 2^{O(\sqrt{\log n \log\log n})}$	Det.

[6] Combining the recent result of [22] with [12] seems to improve the deterministic spanner construction to $\mathsf{poly}\log n$ rounds, and thus provide $\widetilde{O}(\lambda)$-round algorithm for λ-sparse certificates.
[7] I.e., a subgraph $H \subseteq G$ satisfying that H is λ-vertex connected iff G is λ-vertex connected.

Graph Notation. For a subgraph $G' \subseteq G$ and $u, v \in V(G')$, let $\pi(u, v, G')$ be the unique[8] shortest-path in G'. When G' is clear from the context, we may omit it and simply write $\pi(u, v)$. For $u, v \in G$, let $\text{dist}(u, v, G)$ be the length of the shortest u-v path in G. For a vertex pair s, t and subgraph $G' \subseteq G$, let $\lambda(s, t, G')$ be the s-t cut in G'.

The Communication Model. We use a standard message passing model, the CONGEST model [19], where the execution proceeds in synchronous rounds and in each round, each node can send a message of size $O(\log n)$ to each of its neighbors.

2 Exact Computation of Small Cuts

Throughout, we consider unweighted multigraphs with diameter D, and (edge or vertex) connectivity at most $\lambda = O(1)$. Before presenting the algorithm we start by considering the following simpler task.

Warm Up: Cut Verification. In the cut verification problem, one is given a subset of edges E' where $|E'| \leq \lambda$, it is then required to test if $G \setminus E'$ is connected. As we will see there is a simple algorithm for this problem which is based on the following key lemma.

▶ **Lemma 6.** *Consider a D-diameter unweighted graph $G = (V, E)$ with maximum degree Δ. (1) If $u, v \in V$ are λ-edge connected*[9] *(i.e., the u-v cut is at least λ) then* $\text{dist}(u, v, G \setminus F) \leq c \cdot \lambda \cdot D$ *for every edge sequence $F \subseteq E$, $|F| \leq \lambda - 1$ for some constant c.*
(2) If $u, v \in V$ are λ-vertex connected then $\text{dist}(u, v, G \setminus F) \leq c \cdot \lambda \cdot \Delta \cdot D$ *for every vertex sequence $F \subseteq V$, $|F| \leq \lambda - 1$.*

Proof. Let T be an arbitrary BFS tree in G of diameter $O(D)$. We begin with (1). Fix a set of faults $F \subseteq E$, $|F| \leq \lambda - 1$, and let $P_{u,v,F}$ be the u-v shortest path in $G \setminus F$. Since u and v are λ-edge connected such path $P_{u,v,F}$ exists. We next bound the length of $P_{u,v,F}$. Consider the forest $T \setminus F$ which has at most λ connected components $C_1, \ldots, C_\ell$. We mark each vertex on $P_{u,v,F}$ with its component ID in the forest $T \setminus F$. Note that all vertices in the same component are connected by a path of length $O(D)$ in $G \setminus F$. We can then traverse the path $P_{u,v,F}$ from u and jump to the last vertex u_1 on the path (close-most to v) that belongs to the component of u. The length of this sub-path is $O(D)$ and using one connecting edge on $P_{u,v,F}$, we move to a vertex belonging to a new component in $T \setminus F$. Overall the path $P_{u,v,F}$ can be covered by λ path segments $P_1, \ldots, P_\ell$ such that the endpoints of each segments are in the *same* component in $T \setminus F$, each neighboring segments P_i and P_{i+1} are connected by an edge from $P_{u,v,F}$. Since each $|P_i| = O(D)$, we get that $|P_{u,v,F}| = O(\lambda \cdot D)$. Claim (2) follows the exact same argument with the only distinction is that when a vertex fails, the BFS tree might break up into $\Delta + 1$ components. Thus, for a subset $F \subseteq V$ of vertices with $|F| \leq \lambda - 1$, the tree T breaks into $O(\lambda \cdot \Delta)$ components. ◀

This lemma immediately implies an $O(\lambda D)$-round solution for the cut verification task: build a BFS tree T from an arbitrary source up to depth $O(\lambda \cdot D)$. Then T is a spanning tree iff E' does not disconnect the graph.

▶ **Corollary 7** (Cut Verification). *Given a set of λ edges E'. One can test if E' is a cut in G using $O(\lambda \cdot D)$ rounds.*

---

[8] Ties are broken is a consistent manner.
[9] Note that we do not require the graph G to be λ connected.

The following definition is useful for the description and analysis of our algorithm.

▶ **Definition 8** ((s,t) connectivity certificate)**.** *Given a graph G with minimum cut λ and a pair of vertices s and t, the (s,t) connectivity certificate is a subgraph $G_{s,t} \subseteq G$ satisfying that s and t are λ-connected in $G_{s,t}$ iff they are λ-connected in G.*

Whereas a-priori the size of the s-t connectivity certificate, measured by the number of edges, might be $\Omega(n)$, as will show later on, it is in fact bounded by $(\lambda D)^{O(\lambda)}$, hence $\mathsf{poly}(D)$ for $\lambda = O(1)$. With this definition in mind, we are now ready to present the minimum cut algorithm.

A poly(D)-Round Randomized Algorithm. The algorithm has two phases. In the first phase, every vertex t computes its (s,t) certificate subgraph $G_{s,t}$ w.r.t. a given fixed source s. In the second phase, each vertex y locally computes its s-t cut in the subgraph $G_{s,t}$, and one of the output λ-size cut is broadcast to the entire network. Throughout, we assume w.l.o.g. that the value of the minimum cut λ is known, since $\lambda = O(1)$ this assumption can be easily removed.

The first phase has $\ell = O((\lambda D)^{2\lambda})$ iterations, or *experiments*. In each iteration i, the algorithm samples a subgraph G_i by including each edge $e \in G$ into G_i independently with probability $p = 1 - 1/(c(\lambda D))$ for some constant c (taken from Lemma 6). For a source vertex s (which is fixed in all iterations), a (truncated) BFS tree B_i rooted in s is computed in G_i up to depth $c \cdot \lambda \cdot D$. Next, every vertex in B_i learns its tree path from s by pipelining these edges downward the tree. This completes the description of an iteration. Let $G_{s,t} = \bigcup_{i=1}^{\ell} \pi(s,t,B_i)$.

In the second phase, every vertex t locally computes its s-t cut in the subgraph $G_{s,t}$. The edges of the minimum cut are those obtained by one of the vertices t whose s-t connectivity in $G_{s,t}$ is at most λ.

Correctness. For the correctness of the algorithm it will be sufficient to show that w.h.p. $G_{s,t}$ is an s-t connectivity certificate for every $t \in V$.

▷ Claim 9. For every t, w.h.p., $G_{s,t}$ is an s-t connectivity certificate.

Proof. Since $G_{s,t} \subseteq G$, it is sufficient to show that s and t are connected in $G_{s,t} \setminus F$ for any subset $F \subset E$, $|F| \leq \lambda$ satisfying that s and t are connected in $G \setminus F$. Fix such a triplet $\langle s,t,F \rangle$ where s and t are connected in $G \setminus F$. An iteration i is *successful* for $\langle s,t,F \rangle$ if

$$\pi(s,t,G \setminus F) \subseteq G_i \ \text{ and } \ F \cap G_i = \emptyset.$$

Note that if iteration i is successful for $\langle s,t,F \rangle$, then the truncated BFS tree B_i contains t as $\mathrm{dist}(s,t,G_i) = |\pi(s,t,G \setminus F)| \leq c \cdot \lambda \cdot D$, where the last inequality is due to Lemma 6(1). In addition, since $F \cap G_i = \emptyset$, it also holds that $\pi(s,t,B_i) \subseteq G_{s,t} \setminus F$.

It remains to show that with probability at least $1 - 1/n^{\Omega(\lambda)}$, every triplet $\langle s,t,F \rangle$ where s and t are connected in $G \setminus F$, has at least one successful iteration. The claim will then follow by applying the union bound over all $n^{2\lambda}$ triplets. Recall that each edge is sampled into G_i independently with probability p. Thus the probability that iteration i is successful for $\langle s,t,F \rangle$ is at least:

$$q = p^{(c \cdot \lambda D)} \cdot (1-p)^{\lambda} = 1/(\lambda \cdot D)^{\lambda}.$$

Since there are ℓ independent experiments, the probability that all of them fail is $(1-q)^{\ell} \leq 1/n^{\Omega(\lambda)}$, the claim follows. ◁

Finally, let t be a vertex such that s and t are not $(\lambda+1)$-connected in G. Thus, by the lemma above, $\lambda(s,t,G_{s,t}) \leq \lambda$ and the minimum cut computation applied locally by vertex t in $G_{s,t}$ outputs a subset F of at most λ edges. We claim that w.h.p. s and t are also not connected in $G \setminus F$. Assume otherwise, then by Lemma 6(1), $\text{dist}(s,t,G \setminus F) \leq c \cdot \lambda \cdot D$, thus by the argument above, w.h.p., there is an iteration i in which an s-t path that does not go through F is taken into $G_{s,t}$, leading to a contradiction that s and t are disconnected in $G_{s,t} \setminus F$.

▶ **Corollary 10.** *For every D-diameter unweighted graph $G=(V,E)$, $\lambda \geq 1$ and vertex pair $s,t \in V$, there exists an (s,t) certificate $G_{s,t} \subseteq G$ of size $(\lambda D)^{O(\lambda)}$.*

Round Complexity. Each of the ℓ iterations takes $O(\lambda \cdot D)$ rounds for computing the truncated BFS tree. Learning the edges along the tree path from the root also takes $O(\lambda \cdot D)$ rounds via pipeline, thus overall the round complexity is $(\lambda \cdot D)^{(c+2)\lambda} = \mathsf{poly}(D)$ for $\lambda = O(1)$.

Extension to Vertex Cuts The algorithm for computing vertex cuts is almost identical and requires minor adaptations. First, instead of having a single source vertex s, we will pick $\lambda+1$ arbitrary sources $s_1, \ldots, s_{\lambda+1}$ and will run an algorithm, which is very similar to the one described above, with respect to each source s_i. Note that since the vertex cut has size λ, then there is at least one vertex cut $V' \subset V$ of size λ that does not contain at least one of the sources s_i. In such a case, our algorithm will find the cut V' when running the below mentioned algorithm w.r.t the source s_i.

The algorithm for each source s_i works in iterations, where each iteration j samples into a subgraph G_j a collection of vertices rather than edges. That is, the subgraph G_j is defined by taking the induced graph on a sample of vertices, where each vertex gets sampled independently with probability $p' = (1 - 1/(c\lambda \cdot \Delta \cdot D))$ (the constant c is taken from Lemma 6(2)). Then a BFS tree B_j rooted at s_i is computed in G_j up to depth $c\lambda \cdot \Delta \cdot D$. Every vertex $v \in B_j$ collects its path from the root. Let $G_{s_i,t}$ be the union of all paths collected for each vertex t. In the second phase, the vertex t computes locally the s_i-t vertex-cut in $G_{s_i,t}$. The analysis is then identical to that of the edge case, where in particular, we get that w.h.p. $G_{s_i,t}$ is the vertex-connectivity certificate for every t.

2.1 Deterministic Min-Cut Algorithms via Universal Sets

Our goal in this section is to derandomize the FT-sampling technique by locally computing explicitly (at each node) a *small* family of graphs $\mathcal{G} = \{G_i \subseteq G\}$ such that in iteration i, the vertices will apply the computation on the graph G_i in the same manner as in the randomized algorithm. Here, however, the graph G_i is not sampled but rather computed locally by all the vertices. The family of subgraphs $\mathcal{G}$ is required to satisfy the following crucial property for $a = c \cdot D \cdot \lambda$ and $b = \lambda$:

For every two disjoint subsets of edges A, B with $|A| \leq a$ and $|B| \leq b$, there exists a subgraph $G_i \in \mathcal{G}$ satisfying that:

$$A \subseteq G_i \quad \text{and} \quad B \cap G_i = \emptyset \ . \tag{1}$$

In our algorithms, the subset B corresponds to a set of edge faults, and A corresponds to an s-t shortest path in $G \setminus B$. Thus, $|B| \leq \lambda$ and by Lemma 6, $|A| = O(\lambda \cdot D)$. We begin with the following observation that follows by the probabilistic method.

▶ **Lemma 11.** *There exists a family of graphs $\mathcal{G} = \{G_i \subseteq G\}$ of size $O(a^{b+1} \cdot \log n)$ that satisfies Eq. (1) for every disjoint $A, B \subseteq E$ with $|A| \leq a$ and $|B| \leq b$.*

Proof. We will show that a random family $\mathcal{G}_R$ with $\ell = O(a^{b+1} \cdot \log n)$ subgraphs satisfies Eq. (1) with non-zero probability. Each subgraph G_i in $\mathcal{G}_R$ is computed by sampling each edge in G into G_i with probability of $p = (1 - 1/a)$.

The probability that G_i satisfies Eq. (1) for a fixed set A and B of size at most a and b (respectively) is $q = p^a \cdot (1-p)^b = 1/a^b$. The probability that none of the subgraph G_i satisfy Eq. (1) for A, B is at most $(1-q)^{a^{b+1} \cdot \log n} \leq 1/n^{3a}$. Thus, by doing a union bound over all n^{2a} possible subsets of A, B, we get that $\mathcal{G}_R$ satisfies Eq. (1) for all subsets with positive probability. The lemma follows. ◀

Lemma 11 already implies a deterministic minimum cut algorithm with $\mathsf{poly}(D)$ rounds, in case where nodes are allowed to perform *unbounded* local computation. Specifically, let every node compute locally, in a brute force manner, the family of graphs $\mathcal{G} = \{G_i \subseteq G\}$ of size $a^{b+1} \cdot \log n$. In each iteration i of the minimum-cut computation, nodes will use the graph $G_i \in \mathcal{G}$ to compute the truncated BFS tree, and collect their tree paths in these trees. Although the CONGEST model does allow for an unbounded local computation, it is still quite undesirable. To avoid this, we next describe an explicit *polynomial* construction of the graph family $\mathcal{G}$. This explicit construction is based on stating our requirements in the language of universal sets.

Universal Sets. A family of sets $\mathcal{S} = \{S \subseteq [n]\}$ is (n,k)-*universal* if every subset $S' \subseteq [n]$ of $|S'| = k$ elements is *shattered* by $\mathcal{S}$. That is, for each of the 2^k subsets $S'' \subseteq S'$ there exists a set $S \in \mathcal{S}$ such that $S' \cap S = S''$. Using linear codes, one can compute (n,k)-universal sets with $n^{O(k)}$ subsets. Alon [1] showed an explicit construction of size $2^{O(k^4)} \log n$ using the Justesen-type codes constructed by Friedman [8]. In our context, the parameter n corresponds to the number of graph edges, and each subset is a subgraph. The parameter k corresponds to the bound on the length of the path which is $O(\lambda \cdot D)$. Using the existing constructions lead to a family with $2^{\lambda \cdot D}$ subgraphs which is unfortunately super-linear, already for graphs of logarithmic diameter.

A New Variant of Universal Sets. We define a more relaxed variant of universal sets, for which a considerably improved size bounds can be obtained. In particular, for our purposes it is not really needed to fully shatter subsets of size k. Instead, for every set S' of k elements we would like that for every small subset $S'' \subseteq S$, $|S''| \leq b$ (which plays the role of the faulty edges), there will be a set S in the family satisfying that $S' \cap S = S' \setminus S''$. We call this variant FT-universal sets, formally defined as follows.

▶ **Definition 12** (FT-Universal Sets). *For integers n, a, b where $a \leq b \leq n$, a family of sets $\mathcal{S} = \{S \subset [1,n]\}$ is (n,a,b)-*universal *if for every two disjoint subsets $A \subset [1,n]$ and $B \subset [1,n]$ where $|A| \leq a$ and $|B| \leq b$, there exists a set $S \in \mathcal{S}$ such that (1) $A \subseteq S$ and (2) $B \cap S = \emptyset$.*

Our goal is compute a family of (n,a,b)-universal sets of cardinality $O(a^{b+1} \log n)$ in time $\mathsf{poly}(n, a^b)$. Towards that goal we will use the notion of perfect hash functions.

▶ **Definition 13** (Perfect Hash Functions). *For integers n and $k < n$ a family of hash functions $\mathcal{H} = \{h : [n] \to [\ell]\}$ is* prefect *if for every subset $S \subset [1,n]$ for $|S| \leq k$, there exists a function $h \in \mathcal{H}$ such that $h(i) \neq h(j), \quad \forall i,j \in S, i \neq j$.*

▶ **Definition 14** (Almost Pairwise Independence). *A family of functions $\mathcal{H}$ mapping domain $[n]$ to range $[m]$ is ϵ-almost pairwise independent if for every $x_1 \neq x_2 \in [n]$, $y_1, y_2 \in [m]$, we have:* $\Pr[h(x_1) = y_1$ *and* $h(x_2) = y_2] \leq (1+\epsilon)/m^2$.

▶ **Fact 15** ([25]). *For every $\alpha, \beta \in \mathbb{N}$ and $\epsilon \in (0,1)$, one can compute in* $\mathsf{poly}(\alpha \cdot 2^\beta \cdot 1/\epsilon)$ *an explicit family of ϵ-almost pairwise independent hash functions $\mathcal{H}_{\alpha,\beta} = \{h : \{0,1\}^\alpha \to \{0,1\}^\beta\}$ that contains $O(\alpha \cdot 2^\beta)$ functions.*

We next show how to compute a family of (n,k)-perfect hash functions in polynomial time.

▷ Claim 16. One can compute an family of (n,k)-perfect hash functions $\mathcal{H} = \{h : [n] \to [2k^2]\}$ of cardinality $O(k^4 \log n)$ in time $\mathsf{poly}(n,k)$.

Proof. We use Fact 15 with $\alpha = \log n$, $\beta = 4 \log k$, and $\epsilon = 0.1$, to get an ϵ-almost pairwise independent hash function family $\mathcal{H}_{\alpha,\beta} = \{h : \{0,1\}^\alpha \to \{0,1\}^\beta\}$. We now show that this family is perfect for subsets $S \in [1,n]$ of cardinality at most k. Fix a subset $S \in [1,n]$, $|S| \leq k$. By definition, for every $x_1, x_2 \in S$ and $y_1, y_2 \in [2k^2]$,

$$\Pr_{h \in \mathcal{H}}[h(x_1) = y_1 \text{ and } h(x_2) = y_2] \leq (1+\epsilon)/(4k^4).$$

Thus, the probability that a uniformly chosen random function $h \in \mathcal{H}_{a,b}$ collides on S is

$$\begin{aligned} \sum_{x_1 \neq x_2 \in S} \Pr_{h \in \mathcal{H}}[h(x_1) = h(x_2)] &\leq k^2 \cdot \max_{x_1 \neq x_2 \in S} \Pr_{h \in \mathcal{H}}[h(x_1) = h(x_2)] \\ &= k^2 \cdot \max_{x_1 \neq x_2 \in S} \sum_{y \in [2k^2]} \Pr[h(x_1) = h(x_2) = y] \leq 0.3 , \end{aligned} \tag{2}$$

by using the fact that $\Pr_{h \in \mathcal{H}}[h(x_1) = h(x_2) = y] \leq (1+\epsilon)/(4k^4)$ (see Def. 14). We get that there exists $h' \in \mathcal{H}_{\alpha,\beta}$ that has no collisions on S. As this holds for every S, the claim follows. ◁

Equipped with the polynomial construction of families of (n,k)-perfect hash functions, we next show how to compute our universal sets in polynomial time.

▶ **Lemma 17** (Small Universal Sets). *For every set of integers $b < a < n$, one can compute in* $\mathsf{poly}(n, a^b)$, *a family of universal sets $\mathcal{S}_{n,a,b}$ of cardinality $O(a^{b+1} \cdot \log n)$.*

Proof. Set $k = a + b$. We will use Claim 16 to compute an (n,k)-perfect family of hash functions $\mathcal{H} = \{h : [n] \to [2k^2]\}$. For every $h \in \mathcal{H}$ and for every subset $i_1, \ldots, i_b \in [1, 2k^2]$, define:

$$S_{h,i_1,i_2,\ldots,i_b} = \{\ell \in [n] \ \mid \ h(\ell) \notin \{i_1, i_2, \ldots, i_b\}\} .$$

Overall, $\mathcal{S}_{n,a,b} = \{S_{h,i_1,i_2,\ldots,i_b} \ \mid \ h \in \mathcal{H}, i_1, i_2, \ldots, i_b \in [1, 2k^2]\}$.

The size of $\mathcal{S}_{n,a,b}$ is bounded by $|\mathcal{H}| \cdot k^{2b} = O(k^{3b} \cdot \log n)$ as desired. We now show that $\mathcal{S}_{n,a,b}$ is indeed a family of universal sets for n, a, b. Since $\mathcal{H}$ is an (n,k) perfect family of hash functions, for every two disjoint subsets $A, B \subset [n]$, $|A| \leq a$ and $|B| \leq b$, there exists a function h that does not collide on $C = A \cup B$ (since $|C| \leq k$). That is, there exists a function $h \in \mathcal{H}$ such that $h(i) \neq h(j)$ for every $i, j \in C$, $i \neq j$. Thus, letting $B = \{s_1, \ldots, s_b\}$ and $i_1 = h(s_1), \ldots, i_b = h(s_b)$, we have that $h(s'_j) \notin \{i_1, \ldots, i_b\}$ for every $s'_j \in A$. Therefore, the subset $S_{h,i_1,i_2,\ldots,i_b}$ satisfies that $A \subseteq S_{h,i_1,i_2,\ldots,i_b}$ and $B \cap S_{h,i_1,i_2,\ldots,i_b} = \emptyset$. ◀

Deterministic Min-Cut Algorithm. Finally, we describe how to use FT-universal sets to get a $\mathsf{poly}(D)$-round distributed algorithm for exact computation of small cuts. The only randomized part of the algorithm above is in defining the $\ell = \mathsf{poly}(D)$ subgraphs G_i. Instead of sampling these subgraphs, each vertex computes them explicitly and locally. First, we rename all the edges to be in $[1, m]$. This can be easily done in $O(D)$ rounds. Now, each vertex locally computes a family of universal sets for parameters $m, k = O(\lambda \cdot D), q = \lambda$. By Lemma 17, the family $\mathcal{S}$ contains $(\lambda \cdot D)^{\lambda} = \mathsf{poly}(D)$ subsets in $[1, m]$. Each of the sets $S_i \in \mathcal{S}$ will be used as a subgraph G_i in the ith iteration. That is, we iterate over all subsets (subgraphs) in $\mathcal{S}$. In iteration i, all vertices know the set S_i and thus can locally decide which of their incident edges is in G_i. The correctness now follows the exact same line as that of the randomized algorithm.

3 Computation of All Edge Connectivities

Finally, we consider the more general task of computing the edge connectivity of all graph edges up to some constant value λ. For an edge $e = (u, v)$, let $\lambda(e)$ be the u-v edge connectivity in G, that is, the number of edge-disjoint u-v paths in G. By using the recent notion of low-congestion cycle cover [17], we show:

▶ **Lemma 18** (Distributed All Edge Connectivities). *For every D-diameter graph G, there is a randomized distributed algorithm that w.h.p. computes all edge connectivities up to some constant value λ within $2^{O(\sqrt{\log n})} \cdot \mathsf{poly}(D)$ rounds. That is, in the output solution, the endpoints of every edge $e = (u, v)$ know the connectivity $\lambda(e)$ of this edge, as well as a certificate for that connectivity.*

Low Congestion Cycle Covers. A (d, c) cycle cover $\mathcal{C}$ is a collection of cycles of length at most d, such that each edge appears on at least one cycle and at most c cycles. We will use the recent deterministic distributed construction of cycle covers of [18].

▶ **Lemma 19** ([18], Distributed Cycle Cover). *For every bridgeless n-vertex graph $G = (V, E)$ with diameter D, one can compute a (d, c) cycle cover $\mathcal{C}$ with $\mathsf{d} = 2^{O(\sqrt{\log n})} \cdot D$ and $\mathsf{c} = 2^{O(\sqrt{\log n})}$, within $\widetilde{O}(\mathsf{d} \cdot \mathsf{c})$ rounds.*

Combining the lemma above with the centralized construction of nearly-optimal cycle covers of [17], we get:

▶ **Corollary 20** (Distributed Opt. Cycle Cover). *For an n-vertex graph $G = (V, E)$ (not necessarily connected), there is a randomized algorithm* ApproxCycleCover *that given the graph G and parameter D' computes w.h.p. a cycle collection $\mathcal{C}$ such that: (a) every edge e that lies on a cycle C_e in G of length at most D' is covered by a cycle $C' \in \mathcal{C}$ of length $2^{O(\sqrt{\log n})} \cdot |C_e|$, and (b) each edge appears on $2^{O(\sqrt{\log n})}$ cycles. In the output format of the algorithm, every edge e learns all the cycles in $\mathcal{C}$ that go through this edge. The round complexity of Alg.* ApproxCycleCover *is $2^{O(\sqrt{\log n})} \cdot D'$.*

The high level idea of Alg. ApproxCycleCover is based on the notion of neighborhood covers. Roughly speaking, the k neighborhood cover for a graph G is a collection of subgraphs $G_1, \ldots, G_\ell$ such that the following three properties hold: (1) for each vertex v, there is a subgraph G_i that contains its entire k-hop neighborhood, (2) the diameter of each graph G_i is $O(k \log n)$, and (3) each vertex appears on $O(\log n)$ subgraphs. Alg. ApproxCycleCover is obtained by applying the cycle cover algorithm of Theorem 19 on each subgraph G_i in the

k neighborhood cover of G, for every value $k = 2^j$, $j \in \{1, \lceil \log(D') \rceil\}$. This increases the total congestion of the cycles by at most a logarithmic factor. To see why this works, consider an edge e that lies on a cycle C_e of length $|C_e| \leq D'$ in G. Letting $|C_e| \in [2^{j-1}, 2^j]$, we have that C_e is fully contained in one of the subgraphs of a k neighborhood cover for $k = 2^j$. Hence, due to Theorem 19 the edge e is covered by a cycle of length $2^{O(\sqrt{\log n})}|C_e|$ as desired.

From Cycle Covers to Edge Connectivities. The algorithm for computing all the edge connectivities is based on combining the FT-sampling approach with Alg. ApproxCycleCover. In the high level, using the sampling technique, the algorithm attempts to compute not a single cycle for covering an edge $e = (u, v)$, but rather a collection of λ edge-disjoint cycles that covers this edge (i.e., the edge e is the only common edge in these cycles). If it fails in finding these edge disjoint cycles, it deduces that the u-v connectivity is less than λ. In the latter case, it also finds all u-v cuts in G.

Let $D' = 2c \cdot \lambda \cdot D + 1$. The algorithm consists of $\ell = O(\lambda \cdot D^\lambda \log n)$ iterations that we treat as *experiments*. In each experiment i, we sample each edge $e \in E(G)$ into G_i with probability $p = (1 - 1/(D')^\lambda)$, and compute a cycle cover $\mathcal{C}_i$ by applying Alg. ApproxCycleCover on G_i with parameter D'. For every edge $e = (u, v)$, let $G_{u,v} = \bigcup_i \{C \mid e \in C, C \in \mathcal{C}_i\}$ be the union of all cycles that go through e. The nodes u, v compute the edge connectivity of e by *locally* computing the u-v cut in the subgraph $G_{u,v}$. For a pseudo-code of the algorithm see Algorithm 1. By a similar argument to that of Cl. 9, we show:

▷ Claim 21. The subgraph $G_{u,v}$ is a u-v connectivity certificate up to connectivity of λ.

Proof. Let $e = (u, v)$ such that u and v are λ-edge connected. In other words, u and v are $(\lambda - 1)$-connected in $G \setminus \{e\}$. Note that since the diameter of $G \setminus \{e\}$ is at most $2D$, by Lemma 6, for every $F \subseteq E(G) \setminus \{e\}$, $|F| \leq \lambda - 2$, we have that $\text{dist}(u, v, G \setminus (F \cup \{e\})) \leq 2c \cdot \lambda \cdot D$. Therefore, for any $F \subseteq E(G) \setminus \{e\}$, $|F| \leq \lambda - 2$ the subgraph $G \setminus F$ contains a cycle that covers e of length at most $D' = 2c \cdot \lambda \cdot D + 1$.

To show that $G_{u,v}$ is a λ-certificate for such a neighboring pair u, v (that is λ edge connected in G), it is sufficient to show that for every failing of at most $\lambda - 2$ edges F where $e \notin F$, $G_{u,v}$ contains a u-v path in $G_{u,v} \setminus (F \cup \{e\})$. Or in the other words, that $G_{u,v} \setminus F$ contains a cycle that covers e.

Fix a failing set F, where $e \notin F$ and $|F| \leq \lambda - 2$, and u and v are connected in $G \setminus (F \cup \{(u, v)\})$. We say that iteration i is successful for such a triplet $\langle u, v, F \rangle$ if $F \cap G_i = \emptyset$, $(u, v) \in G_i$ and $\pi(u, v, G \setminus (F \cup \{(u, v)\})) \subseteq G_i$. Note that in such a case, since G_i contains a cycle of length at most D' that covers e, Algorithm ApproxCycleCover computes a cycle $C \subseteq G_i$ that covers the edge $e = (u, v)$, and thus a u-v path in $G_i \setminus (F \cup \{e\})$ as desired. It remains to show that w.h.p. every triplet $\langle u, v, F \rangle$ has at least one successful iteration. Since each edge is sampled w.p. p into G_i, the iteration is successful with probability $\Omega(1/D^\lambda)$. By simple application of the Chernoff bound, we get that the probability that a given triplet $\langle u, v, F \rangle$ does not have a successful iteration is at most $1/n^{c' \cdot \lambda}$. Thus by applying the union bound over all $n^{\lambda+2}$ triplets, the claim follows. ◁

The proof of Lemma 18 follows by Cor. 20 and Cl. 21. Note that this algorithm can be made deterministic while keeping the same round complexity, by using the derandomization of the FT-sampling approach from Sec. 2.1 along with the deterministic neighborhood cover construction of [12].

▶ **Lemma 22.** *All edge connectivities, up to a constant λ, can be computed deterministically in $2^{\sqrt{\log n \cdot \log\log n}} \cdot \mathsf{poly}(D)$.*

Proof. The algorithm requires two main adaptations. First the randomized algorithm ApproxCycleCover is made deterministic by using the deterministic construction of neighborhood covers of [12] that uses $2^{\sqrt{\log n \cdot \log\log n}}$ rounds. In the second part, we use the derandomization of the FT-sampling, as in the minimum cut algorithm. Overall, the total round complexity is $2^{O(\sqrt{\log n \cdot \log\log n})} \cdot \mathsf{poly}(D)$. ◀

Algorithm 1 DistEdgeConnec($G = (V, E), \lambda$).

Distributed Computation of λ-edge connected cycle covers.

---

1. For $i = 1$ to $O(\lambda \cdot D)^{2\lambda}$:
   - Sample each edge e into G_i w.p. $p = (1 - 1/D')$.
   - $\mathcal{C}_i \leftarrow$ ApproxCycleCover(G_i, D').
2. $\mathcal{C} = \bigcup_i \mathcal{C}_i$.
3. For every edge $e = (u, v)$, let $G_{u,v} = \{C \in \mathcal{C} \mid e \in C\}$.
4. $\lambda(e) = \mathsf{MinCut}(G_{u,v})$.

---

4 Sparse Connectivity Certificates

Finally, we consider the related problem of computing sparse connectivity certificates. We start by observing that an FT-spanner for the graph is also a connectivity certificate.

Fault Tolerant Spanners. Fault tolerant (FT) spanners [14, 4] are sparse subgraphs that preserve pairwise distances in G (up to some multiplicative stretch) even when several edges or vertices in the graph fails. These spanners have been introduced by Levcopoulos for geometric graphs [14], and later on by Chechik et al. [4] for general graphs.

▶ **Definition 23** (Fault Tolerant Spanners). *For positive integers k, f, an f edge fault-tolerant $(2k-1)$ spanner for an n-vertex graph $G = (V, E)$ is a subgraph $H \subseteq G$ satisfying that $\mathrm{dist}(u, v, H \setminus F) \leq (2k-1)\mathrm{dist}(u, v, G \setminus F)$ for every $u, v \in V$ and $F \subseteq E$, $|F| \leq f$. Vertex fault-tolerant spanners are defined analogously where the fault $F \subset V$.*

Chechik et al. [4] gave a generic algorithm for computing these spanners against edge failures.

▶ **Fact 24** ([4]). *Let $\mathcal{A}$ be an algorithm for computing the standard (fault-free) $(2k-1)$ spanner with $O(n^{1+1/k})$ edges and time t. Then one can compute f edge fault-tolerant $(2k-1)$ spanners with $O(f \cdot n^{1+1/k})$ edges in time $O(f \cdot t)$.*

Dinitz and Krauthgamer provided a similar transformation for vertex faults that is based on the FT-sampling technique, see Thm. 2.1 of [6].

Certificates from Fault Tolerant Spanners. The relation between FT-spanners and connectivity certificates is based on the following observation.

▶ **Observation 25.** *An f* edge *(resp., vertex) FT spanner $H \subseteq G$ is a certificate for the f* edge *(resp., vertex) connectivity of the graph.*

Proof of Observation 25. Consider a λ-edge connected graph G, and let H be an f-FT-spanner for G with $f = \lambda - 1$. We show that H is λ-edge connected, by showing that for every vertex pair s, t and a subset of F edge faults, $|F| \leq f$, there exists an s-t path in $H \setminus F$. Let P be an s-t path in $G \setminus F$. Since G is λ-connected, such a path exists. For every edge $e = (u, v) \in P \setminus H$, it holds that $\text{dist}(u, v, H \setminus F) \leq 2k - 1$, thus in particular, every neighboring pair u, v on P are connected in $H \setminus F$, the claim follows. The proof of λ-vertex connected graphs works in the same manner. ◀

Since spanners with logarithmic stretch have linear size, fault tolerant spanners with logarithmic stretch are sparse connectivity certificates. By using the existing efficient (in fact local) distributed algorithms for spanners, we get:

▶ **Observation 26.** *(1) There exists a randomized algorithm for computing a sparse λ-edge certificate with $O(\lambda n)$ edges within $O(\lambda \cdot \log^{1+o(1)} n)$ rounds, w.h.p.; (2) There exists a randomized algorithm for computing a λ-vertex certificate for the λ-vertex connectivity with $O(\lambda^2 \cdot \log n)$ edges within $O(\lambda^3 \cdot \log^{2+o(1)} n)$ rounds, w.h.p.;*

Proof. Claim (1) follows by combining the ultra-sparse spanners construction of Pettie [20] with Fact 24. To obtain sparse certificates, we use Fact 24 with algorithm $\mathcal{A}$ taken to be the ultra-sparse algorithm by Pettie [20]. This algorithm computes an $O(2^{\log^* n} \log n)$-spanner H with $O(n)$ edges within $O(\log^{1+o(1)} n)$ rounds. By taking λ disjoint copies of such spanner, constructed sequentially one after the other, we obtain a certificate with $O(\lambda \cdot n)$ edges. In the same manner, Claim (2) follows by plugging the algorithm of Pettie [20] in the meta algorithm for FT-spanners against vertex faults by Dinitz and Krauthgamer (Thm. 2.1 in [6]). ◀

While Obs. 26 gives efficient algorithm when $\lambda = O(1)$, it is less efficient for large connectivity values. In the next lemma we apply the edge sampling technique of Karger [13] to omit the dependency in λ in the round complexity. Ideas along this line appear in [10].

▶ **Lemma 27.** *For every $\lambda = \Omega(\log n)$, and $\epsilon \in [0, 1]$, given an λ-edge connected graph G, one can compute, w.h.p., an $(1 - \epsilon)\lambda$-edge sparse certificate within $O(1/\epsilon^2 \cdot \log^{2+o(1)} n)$ rounds.*

Proof. We restrict attention to $\lambda = \Omega(\log n)$, as otherwise, the lemma follows immediately from Obs. 26. The key idea for omitting the dependency in λ is by randomly decomposing the graph into spanning subgraphs each with connectivity $\min\{\lambda, \Theta(\log n/\epsilon^2)\}$ using random edge sampling, and to run the algorithm of Obs. 26 on each of the subgraphs. We randomly put each edge of G in one of μ subgraphs $G_1, \ldots, G_\mu$ for $\mu = \lceil \lambda \cdot \epsilon^2/(20 \log n) \rceil$. Karger [13] showed that each subgraph G_i has edge-connectivity in $(1 \pm \epsilon)\lambda/\mu$ with high probability. In addition, the summation of the edge-connectivities $\lambda_1, \ldots, \lambda_\mu$ of the subgraphs $G_1, \ldots, G_\mu$ is at least $\lambda(1 - \epsilon)$. The algorithm then computes a λ'-edge sparse certificate H_i for $\lambda' = (1 - \epsilon)\lambda/\mu$ in each G_i subgraph simultaneously. The output certificate H is the union of all H_i subgraphs.

We next analyze the construction, and start with round complexity. Since the G_i subgraphs are edge disjoint, applying the algorithm of Obs. 26 takes $O(\lambda' \cdot \log^{1+o(1)} n)$ rounds which is $O(\log^{2+o(1)} n)$ rounds. The edge bound follows immediately as $|E(H)| = c \cdot \mu \cdot \lambda' n = O(\lambda n)$ as required. It remains to show that H is indeed a $(1 - \epsilon)\lambda$-edge connectivity certificate. Consider a pair of vertices s, t, and a sequence of at most $(1 - \epsilon)\lambda$ edge faults F. We will show that s and t are connected in $H \setminus F$. Since the μ subgraphs are edge-disjoint, there must be a subgraph G_i containing at most $\lambda' = (1 - \epsilon)\lambda/\mu$ of the faults. Let $F_i = F \cap G_i$. Since G_i is λ'-edge connected, s and t are connected in $G_i \setminus F_i$. Since H_i is a λ'-edge certificate of G_i, it also holds that s and t are connected in $H_i \setminus F_i$ and thus also in $H \setminus F$ (i.e., as by definition, $(F \setminus F_i) \cap H_i = \emptyset$). The claim follows. ◀

References

1. Noga Alon. Explicit construction of exponential sized families of k-independent sets. *Discrete Mathematics*, 58(2):191–193, 1986.
2. Noga Alon, Shiri Chechik, and Sarel Cohen. Deterministic Combinatorial Replacement Paths and Distance Sensitivity Oracles. In *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece.*, pages 12:1–12:14, 2019.
3. Keren Censor-Hillel and Michal Dory. Fast Distributed Approximation for TAP and 2-Edge-Connectivity. In *21st International Conference on Principles of Distributed Systems, OPODIS 2017, Lisbon, Portugal, December 18-20, 2017*, pages 21:1–21:20, 2017.
4. Shiri Chechik, Michael Langberg, David Peleg, and Liam Roditty. Fault tolerant spanners for general graphs. *SIAM Journal on Computing*, 39(7):3403–3423, 2010.
5. Mohit Daga, Monika Henzinger, Danupon Nanongkai, and Saranurak. Distributed Edge Connectivity in Sublinear Time. In *STOC*, 2019.
6. Michael Dinitz and Robert Krauthgamer. Fault-tolerant spanners: better and simpler. In *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 169–178. ACM, 2011.
7. Michal Dory. Distributed Approximation of Minimum k-edge-connected Spanning Subgraphs. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018, Egham, United Kingdom, July 23-27, 2018*, pages 149–158, 2018.
8. J Friedman. Constructing O (n log n) size monotone formulae for the k-th elementary symmetric polynomial of n Boolean variables. In *Foundations of Computer Science, 1984. 25th Annual Symposium on*, pages 506–515. IEEE, 1984.
9. Mohsen Ghaffari. *Improved Distributed Algorithms for Fundamental Graph Problems*. PhD thesis, MIT, USA, 2017. URL: `https://groups.csail.mit.edu/tds/papers/Ghaffari/PhDThesis-Ghaffari.pdf`.
10. Mohsen Ghaffari and Fabian Kuhn. Distributed minimum cut approximation. In *International Symposium on Distributed Computing*, pages 1–15. Springer, 2013.
11. Mohsen Ghaffari and Fabian Kuhn. Distributed Minimum Cut Approximation. *arXiv preprint*, 2013. `arXiv:1305.5520`.
12. Mohsen Ghaffari and Fabian Kuhn. Derandomizing Distributed Algorithms with Small Messages: Spanners and Dominating Set. In *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018*, pages 29:1–29:17, 2018.
13. David R Karger. Random sampling in cut, flow, and network design problems. *Mathematics of Operations Research*, 24(2):383–413, 1999.
14. Christos Levcopoulos, Giri Narasimhan, and Michiel Smid. Efficient algorithms for constructing fault-tolerant geometric spanners. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 186–195. ACM, 1998.
15. Hiroshi Nagamochi and Toshihide Ibaraki. A linear-time algorithm for finding a sparsek-connected spanning subgraph of ak-connected graph. *Algorithmica*, 7(1-6):583–596, 1992.
16. Danupon Nanongkai and Hsin-Hao Su. Almost-tight distributed minimum cut algorithms. In *International Symposium on Distributed Computing*, pages 439–453. Springer, 2014.
17. Merav Parter and Eylon Yogev. Low Congestion Cycle Covers and Their Applications. *SODA*, 2019.
18. Merav Parter and Eylon Yogev. Optimal Short Cycle Decomposition in Almost Linear Time. *ICALP*, 2019.
19. David Peleg. *Distributed Computing: A Locality-sensitive Approach*. SIAM, 2000.
20. Seth Pettie. Distributed algorithms for ultrasparse spanners and linear size skeletons. *Distributed Computing*, 22(3):147–166, 2010. `doi:10.1007/s00446-009-0091-7`.
21. David Pritchard and Ramakrishna Thurimella. Fast computation of small cuts via cycle space sampling. *ACM Transactions on Algorithms (TALG)*, 7(4):46, 2011.
22. Václav Rozhon and Mohsen Ghaffari. Polylogarithmic-Time Deterministic Network Decomposition and Distributed Derandomization. *arXiv preprint*, 2019. `arXiv:1907.10937`.

23 Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. Distributed verification and hardness of distributed approximation. *SIAM Journal on Computing*, 41(5):1235–1265, 2012.

24 Ramakrishna Thurimella. Sub-linear distributed algorithms for sparse certificates and biconnected components. *Journal of Algorithms*, 23(1):160–179, 1997.

25 Salil P. Vadhan. Pseudorandomness. *Foundations and Trends® in Theoretical Computer Science*, 7(1–3):1–336, 2012. doi:10.1561/0400000010.

26 Xiao Fan Wang and Guanrong Chen. Complex networks: small-world, scale-free and beyond. *IEEE circuits and systems magazine*, 3(1):6–20, 2003.

27 Oren Weimann and Raphael Yuster. Replacement paths and distance sensitivity oracles via fast matrix multiplication. *ACM Transactions on Algorithms (TALG)*, 9(2):14, 2013.

Monotonically Relaxing Concurrent Data-Structure Semantics for Increasing Performance: An Efficient 2D Design Framework

Adones Rukundo
Chalmers University of Technology Department of Computer Science and Engineering, Sweden
adones@chalmers.se

Aras Atalar
Chalmers University of Technology Department of Computer Science and Engineering, Sweden
aaras@chalmers.se

Philippas Tsigas
Chalmers University of Technology Department of Computer Science and Engineering, Sweden
tsigas@chalmers.se

Abstract

There has been a significant amount of work in the literature proposing semantic relaxation of concurrent data structures for improving scalability and performance. By relaxing the semantics of a data structure, a bigger design space, that allows weaker synchronization and more useful parallelism, is unveiled. Investigating new data structure designs, capable of trading semantics for achieving better performance in a monotonic way, is a major challenge in the area. We algorithmically address this challenge in this paper.

We present an efficient, lock-free, concurrent data structure design framework for *out-of-order* semantic relaxation. We introduce a new two dimensional algorithmic design, that uses multiple instances of a given data structure. The first dimension of our design is the number of data structure instances operations are spread to, in order to benefit from parallelism through disjoint memory access; the second dimension is the number of consecutive operations that try to use the same data structure instance in order to benefit from data locality. Our design can flexibly explore this two-dimensional space to achieve the property of monotonically relaxing concurrent data structure semantics for better performance within a tight deterministic *relaxation* bound, as we prove in the paper.

We show how our framework can instantiate lock-free *out-of-order* queues, stacks, counters and dequeues. We provide implementations of these *relaxed* data structures and evaluate their performance and behaviour on two parallel architectures. Experimental evaluation shows that our two-dimensional design significantly outperforms the respected previous proposed designs with respect to scalability and performance. Moreover, our design increases performance monotonically as relaxation increases.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis; Theory of computation → Design and analysis of algorithms; Theory of computation → Concurrency; Theory of computation → Concurrent algorithms; Computing methodologies → Concurrent algorithms

Keywords and phrases Lock Free, Concurrency, Semantics Relaxation, Data Structures

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.31

Related Version An extended version of the paper is available at `https://arxiv.org/abs/1906.07105`.

Funding SIDA/Bright Project (317) under the Makerere-Sweden bilateral research programme 2015-2020, Mbarara University of Science and Technology, and, Swedish Research Council (Vetenskapsrådet) Under Contract No.: 2016-05360; "Models and Techniques for Energy-Efficient Concurrent Data Access Designs".

© Adones Rukundo, Aras Atalar, and Philippas Tsigas;
licensed under Creative Commons License CC-BY
33rd International Symposium on Distributed Computing (DISC 2019).
Editor: Jukka Suomela; Article No. 31; pp. 31:1–31:15

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Concurrent data structures allow operations to access the data structure concurrently, which require synchronised access to guarantee consistency with respect to their sequential semantics [9, 10]. The synchronisation of concurrent accesses is generally achieved by guaranteeing some notion of atomicity, where, an operation appears to occur at a single instant between its invocation and its response. A concurrent data structure is typically designed around one or more synchronisation *access* points, from where threads compute, consistently, the current state of the data structure. Synchronisation is vital to achieving consistency and cannot be eliminated [5]. Whereas this is true, synchronization might generate *contention* in memory resources hurting scalability and performance.

The necessity of reducing contention at the synchronisation *access* points, and consequently improving scalability, is and has been a major focus for concurrent data structure researchers. Techniques like; elimination [18, 31], combining [32], dynamic elimination-combining [6] and back-off strategies have been proposed as ways to improve scalability. To address, in a more significant way, the challenge of scalability bottlenecks of concurrent data structures, it has been proposed that the semantic legal behaviour of data structures should be extended [29]. This line of research has led to the introduction of an extended set of weak semantics including; weak internal ordering, weakening consistency and semantic *relaxation*.

One of the main definition of semantic *relaxation* proposed and used in the literature is *k-out-of-order* [1, 2, 15, 19, 24, 26, 33, 35]. *k-out-of-order* semantics allow operations to occur out of order within a given k bound, e.g. a pop operation of a *k-out-of-order* stack can remove any item among the k topmost stack items. By allowing a *Pop* operation to remove any item among the k topmost stack items, the semantics do not anymore impose a single *access* point. Thus, by *relaxing* the stack semantics, we allow for potentially more efficient stack designs with reduced synchronisation overhead, which is the motivation for concurrent data structure semantics *relaxation*.

Relaxation can be exploited to achieve improved parallelism by increasing the number of disjoint *access* points, or by increasing thread local data processing. Disjoint *access* is popularly achieved by distributing operations over multiple instances of a given data structure [2, 14, 15, 24]. On the other hand, the locality is generally achieved through binding single thread *access* to the same memory location for specific operations [13, 14, 35].

In this paper, we introduce an efficient two-dimensional algorithmic design framework, that uses multiple instances (*sub-structures*) of a given data structure as shown in Figure 1. The first dimension of the framework is the number of *sub-structures* operations are spread to, in order to benefit from parallelism through disjoint *access* points; the second dimension

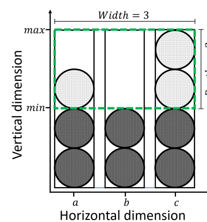


Figure 1 An illustration of our 2D design using a Stack as an example. There are three *sub-stacks* a, b and c. k is proportional to the area of the green dashed rectangle in which operations are bounded to occur. a can be used for both *Push* and *Pop*. b can be used for *Push* but not for *Pop*. c can be used for *Pop* but not for *Push*.

is the number of consecutive operations that can occur on the same *sub-structure* in order to benefit from data locality. We use two parameters to control the dimensions; *width* for the first dimension (horizontal) and *depth* for the second dimension (vertical).

A thread can operate on a given *sub-structure* for as long as a set of conditions hold (*validity*). Validity can be that valid *sub-structures* do not exceed (*max*) or go below (*min*), a given operation count threshold, as depicted by the dashed green rectangle in Figure 1. Validity conditions make *sub-structures* valid or invalid for a given operation. This implies that threads have to search for a valid *sub-structure*, increasing operation cost (latency). Our framework overcomes this challenge by limiting the number of *sub-structures* and allowing a thread to operate on the same *sub-structure* consecutively for as long as the validity conditions hold. *Max* and *min* can be updated if there are no valid *sub-structures*. We show algorithmically that the validity conditions provide for an efficient, tenable and tunable *relaxation* behaviour, described by tight deterministic *relaxation* bounds.

Our design framework can be used to extend existing lock-free data structure algorithms to derive *k-out-of-order* semantics for the given data structure. This can be achieved with minimal modifications to the data structure algorithm as we later show in this paper. Using our framework, we extend existing lock-free algorithms to derive lock-free *k-out-of-order* stacks, queues, dequeue and counters. Detailed implementation, correctness and performance analysis is also provided. Experimental evaluation shows that the derived data structures significantly outperform all previous data structure implementations of same category.

The rest of the paper is structured as follows. In Section 2 we discuss literature related to this work. We present the $2D$ framework and derived 2D algorithms in Section 3. We prove correctness and linearization bounds in Section 4. An experimental evaluation is discussed in Section 5 and the paper concludes in Section 6.

2 Related Work

Recently, data structure semantic *relaxation* has attracted the attention of researchers, as a promising direction towards improving concurrent data structures' scalability [19, 29, 33]. It has also been shown that small changes on the semantics of a data structure can have a significant effect on the computation power of the data structure [30]. The interest in semantic *relaxation* is largely founded on the ease of use and understanding. One of the main definition of semantic *relaxation* proposed and used is *k-out-of-order*.

Using the *k-out-of-order* definition, a segmentation technique has been proposed in [1], later revisited in [19] realizing a *relaxed* Stack (*k-Stack*) and FIFO Queue (*Q-segment*) with *k-out-of-order* semantics. The technique involves a linked-list of memory segments with k number of indexes on which an item can be added or removed. The stack items are accessed through the topmost segment, whereas the queue has a tail and head segment from which *Enqueue* and *Dequeue* can occur respectively. Segments can be added and removed. *Relaxation* is only controlled through varying the number of indexes per segment. As discussed in Section 1, increasing the number of indexes increases operation latency and later becomes a performance bottleneck. This limits the performance benefits of the technique to a small range of *relaxation* values.

Also, load balancing together with multiple queue instances (*sub-queues*) has been used to design a *relaxed* FIFO queue (*lru*) with *k-out-of-order* semantics [15]. Each *sub-queue* maintains two counters, one for *Enqueue* another for *Dequeue*, while two global counters, one for *Enqueue* another for *Dequeue* maintain the total #operations for all *sub-queues*. The global counters are used to calculate the expected #operations on the last-recently-used

sub-queue. Threads can only operate on the least-recently-used *sub-queue*. This implies that for every operation threads must synchronise on the global counter, making it a sequential bottleneck. Moreover, threads have to search for the last-recently-used *sub-queue* leading to latency increase. Randomisation has also been used to balance load between multiple *sub-structures*, leading to *relaxed* designs such as MultiQueues and MultiCounters [2, 24].

The proposed *relaxation* techniques, mentioned above, apply *relaxation* in one dimension, i.e, increase disjoint *access* points to improve parallelism and reduce contention. However, this also increases operation latency due to increased *access* points to select from. Without a remedy to this downside, the proposed techniques cannot provide monotonic *relaxation* for better performance. Other *relaxed* data structures studied in the literature include priority queues [4, 24, 35]. Apart from semantic *relaxation*, other design strategies for improving scalability have been proposed including; elimination [6, 18, 23, 31], combining [32], internal weak ordering [11], and local linearizability [14]. However, these strategies have not been designed to provide bounded out of order semantic *relaxation*.

Elimination implements a collision path on which different concurrent operations try to collide and cancel out, otherwise, they proceed to access the central structure [18]. Combining, on the other hand, allows operations from multiple threads to be combined and executed by a single thread without the other threads contending on the central structure [12, 17]. However, their performance depends on the specific workload characteristics. Elimination mostly benefits symmetric workloads, whereas combining mostly benefits asymmetric workloads. Furthermore, the central structure sequential bottleneck problem still persists.

Weak internal ordering has been proposed and used to implement a timestamped stack (*TS-Stack*) [11], where *Push* timestamps each pushed item to mark the item's precedence order. Each thread has its local buffer on which it performs *Push* operations. However, *Pop* operations pay the cost of searching for the latest item. In the worst case, *Pop* operations might contend on the same latest item if there are no concurrent *Push* operations. This leads to search retries, especially for workloads with higher *Pop* rates than *Push* ones.

Local linearizability has also been proposed for concurrent data structures such as; FIFO queues and Stacks [14]. The technique relies on multiple instances of a given data structure. Each thread is assigned an instance on which it locally linearizes all its operations. Operations: *Enqueue* (FIFO queue) or *Push* (Stack) occur on the assigned instance for a given thread, whereas, *Dequeue* or *Pop* can occur on any of the available instances. With *Dequeue* or *Pop* occurring more frequently, contention quickly builds as threads try to access remote buffers. The threads also lose the locality advantage while accessing remote buffers, cancelling out the caching advantage especially for single access data structures such as the Stack [7, 16, 28].

3 2D Framework

In this section, we describe our 2*D* design framework and show how it can be used to extend existing data structure designs to derive *k-out-of-order relaxed* semantics. Such data structures include; stacks, FIFO queues, counters and dequeues.

The 2D framework uses multiple copies (*sub-structures*) of the given data structure as depicted in Figure 1. Threads can operate on any of the *sub-structures* following the fixed maximum *max* and minimum *min* operation count threshold. Herein, *operation* refers to the process that updates the data structure state by adding (*Put*) or removing (*Get*) an item (*Push* and *Pop* respectively for the stack example). Each *sub-structure* holds a counter (*sub-count*) that counts the number of local successful operations.

Algorithm 1 Window Coupled (2Dc).

```
1  Struct Descriptor Des
2  |    *item;
3  |    count;                                    ▷ sub-count
4  |    version;
5  Struct Window Win
6  |    max;
7  |    version;
8  Function Window(Op,index,cont)
9  |    IndexSearch = Random = notempty=0; LWin = Win;
10 |    if cont==True then
11 |    |    index=RandomIndex(); cont=False;
12 |    end
13 |    while True do
14 |    |    if IndexSearch == width then
15 |    |    |    SHIFTWINDOW();
16 |    |    end
17 |    |    Des = Array[index];                   ▷ Read descriptor
18 |    |    if Op == put ∧ Des.count < Win.max then
19 |    |    |    return {Des,index};
20 |    |    else if Op == get ∧ Des.count > (Win.max - depth) then
21 |    |    |    return {Des,index};
22 |    |    else if LWin == Win then
23 |    |    |    HOP();
24 |    |    else
25 |    |    |    LWin = Win; IndexSearch = 0;
26 |    |    end
27 |    end
28 Macro SHIFTWINDOW()
29 |    if Op == get ∧notempty==0 then
30 |    |    return {Des,index};
31 |    end
32 |    if LWin == Win then
33 |    |    if Op == put then
34 |    |    |    NWin.max = LWin.max + ShiftUp;
35 |    |    else if Op == get ∧ Win.max > depth then
36 |    |    |    NWin.max = LWin.max - ShiftDown;
37 |    |    end
38 |    |    NWin.version = LWin.version + 1;
39 |    |    CAS(Win,LWin,NWin);
40 |    end
41 |    LWin = Win; IndexSearch = 0;
42 Macro HOP()
43 |    if Random<2 then
44 |    |    index=RandomIndex(); Random+=1;
45 |    else
46 |    |    if Op == get ∧ Des.item!=NULL then
47 |    |    |    notempty=1;
48 |    |    end
49 |    |    if index == width - 1 then
50 |    |    |    index=0;
51 |    |    else
52 |    |    |    index += 1;
53 |    |    end
54 |    |    IndexSearch += 1;
55 |    end
```

A combination of *max*, *min* and *#sub-structures*, form a logical count period, we refer to it as *Window*, depicted by the dashed green rectangle in Figure 1. *Window* defines the maximum (Win_{max}) and minimum (Win_{min}) operation count threshold for all *sub-structures*, for a given period. This implies that, for a given period, a *sub-structure* can be valid or invalid as exemplified in Figure 1, and a *Window* can be full or empty. The *Window* is *full* if all *sub-structures* have maximum operations (*sub-count* $= Win_{max}$), *empty*, if all *sub-structures* have minimum operations (*sub-count* $= Win_{min}$). The *Window* is defined by two parameters; *width* and *depth*. *width* $=$ *#sub-structures*, and *depth* $= Win_{max} - Win_{min}$.

To validate a *sub-structure*, its *sub-count* is compared with Win_{min} or Win_{max}; either *sub-count* $\geq Win_{min}$ or *sub-count* $< Win_{max}$. If the given *sub-structure* is invalid, the thread has to *hop* to another *sub-structure* until a valid *sub-structure* is found (validity is operation specific as we discuss later). If a thread cannot find a valid *sub-structure*, then, the *Window* is either full or empty. The thread will then, either increment or decrement both Win_{min} and Win_{max}, the process we refer to as, *Window shifting*. A *Window* can $shift$ up or down, and is controlled by $shift^{up}$ or $shift_{down}$ values respectively, where, $0 < shift^{up}, shift_{down} \leq$ *depth*. Win_{min} and Win_{max} can only be incremented or decremented by a given $shift$ value. $shift^{up}$ and $shift_{down}$ can be configured differently to optimize for different workloads.

We define two types of *windows*: *WinCoupled* (*2Dc*) and *WinDecoupled* (*2Dd*).

WinCoupled: couples both *Put* and *Get* to share the same *Window* and *sub-count* for each *sub-structure*. A successful *Put* increments whereas, a successful *Get* decrements the given *sub-count*. On a full *Window*, *Put* increments Win_{max} *shifting* the *Window* up ($shift^{up}$), whereas, on an empty *Window*, *Get* decrements Win_{max}, *shifting* the *Window* down ($shift_{down}$). *WinCoupled* resembles elimination [18], only that here, we cancel out operation counts for matching *Put* and *Get* on the same *sub-structure* within the same *Window*. Just like elimination reduces joint *access* updates, *WinCoupled* reduces *Window* $shift$ updates.

WinDecoupled: decouples *Put* and *Get* and assigns them independent *windows*. Also, an independent *sub-count* is maintained for *Put* or *Get*, on each *sub-structure*. Unlike *WinCoupled*, both operations always increment their respective *sub-count* on a successful operation and Win_{max} on a full *Window*. This implies that both *sub-count* and *Window* counters are always increasing.

Data structures such as FIFO queues with disjoint *access* for *Put* and *Get*, can benefit more from the *WinDecoupled* disjoint *Window* design. Whereas, data structures such as stacks with joint *access*, can benefit more from the *WinCoupled* operation count cancelling design. Here, we present *WinCoupled* due to its interesting operation count cancelling and refer the reader to our extended version [27] for the *WinDecoupled* presentation.

In Algorithm 1, we present the algorithmic steps for *WinCoupled*. Recall, *width* = #*sub-structures* and *depth* = $Win_{max} - Win_{min}$. Each *sub-structure* is uniquely identified by an index, which holds information including a pointer to the *sub-structure*, *sub-count* counter, and a version number (line 1-4). The version number is to avoid ABA related issues. Using a wide `CAS`, we update the index information in a single atomic step.

To perform an operation, the thread has to search and select a valid *sub-structure* within a *Window* period. Starting from the search start index, the thread stores a copy of the *Window* locally (line 9) which is used to detect *Window* $shifts$ while searching (line 22,32). During the search, the thread validates each *sub-structure* count against Win_{max} (line 18,20). If no valid *sub-structure* is found, Win_{max} is updated atomically, $shifting$ the *Window* up or down (line 39). *Put* increments Win_{max} to $shift$ the *Window* up (line 34), whereas, *Get* decrements Win_{max} to $shift$ the *Window* down (line 36). Before *Window* $shifting$ or index *hopping*, the thread must confirm that the *Window* has not yet shifted (line 32 and 22 respectively). For every *Window* shift during the search, the thread restarts the search with the new *Window* values (line 25,41).

If a valid index is selected, the respective descriptor state and index are returned (line 19,21). The thread can then proceed to try and operate on the given *sub-structure* pointed to by the index descriptor. As an *emptiness check*, the *Window* search can only return an empty *sub-structure* (line 29), if during the search, all *sub-structures* where empty ($NULL$ pointer). Using the *Window* parameters, *width*, and *depth*, we can tightly bound the *relaxation* behaviour of derived 2D data-structures as discussed later in Section 4.

3.1 Deriving 2D Data structures

Our framework can be used to extend existing algorithms to derive *k-out-of-order* data structures. Using *WinCoupled* we derive a *2Dc-Stack* and a *2Dc-Counter*, whereas by using *WinDecoupled*, we derive a *2Dd-Stack*, a *2Dd-Queue*, a *2Dd-Deque* and a *2Dd-Counter*. The base algorithms include but not limited to; Treiber's stack [34], MS-queue [21] and Deque [20] for Stack, FIFO Queue and Deque respectively. As an example, we shall discuss the *2Dc-Stack* due to its simplicity and refer the reader to our extended version [27] for the other algorithmic implementations.

Algorithm 2 *2Dc-Stack.*

```
1  Function Push(NewItem)
2      while True do
3          {Des,Index} = Window(push,index,cont);
4          NewItem.next = Des.item;
5          NDes.item = NewItem;
6          NDes.count = Des.count + 1;
7          NDes.version = Des.version;
8          if CAS(Array[Index],Des,NDes) then
9              return 1;
10         else
11             cont=True;
12         end
13     end

14 Function Pop()
15     while True do
16         {Des,Index}=Window(pop,index,cont);
17         if Des.item != NULL then
18             NDes.item = Des.item.next;
19             NDes.count = Des.count - 1;
20             NDes.version = Des.version;
21             if CAS(Array[Index],Des,NDes) then
22                 return Des.item;
23             else
24                 cont=True;
25             end
26         else
27             return Null;
28         end
29     end
```

As depicted in Algorithm 2, a stack has two operations: *Push* that adds an item and *Pop* that removes an item from the stack. *2Dc-Stack* is composed of multiple lock-free *sub-stacks*. Each *sub-stack* is implemented according to the Treiber's stack design, modified only to fit the *Window* design. The stack head is modified to a descriptor containing the top item pointer, operation count, and descriptor version. Note that, the descriptor is updated in a single atomic step using a wide `CAS` (line 8,21), the same way as in the Treiber's stack.

To perform an operation, a given thread obtains a *sub-stack* by performing a *Window* search (line 3,16). The thread then prepares a new descriptor based on the existing descriptor at the given index (line 4-7,18-20). Using a `CAS`, the thread tries to atomically swap the existing descriptor with the new one (line 8,21). If the `CAS` fails, the thread sets the contention indicator to true (line 11,24) and restart the *Window* search.

A successful *Push* increments whereas a *Push* decrements the operation count by one (line 6,19). Also, the topmost item pointer is updated. At this point, a *Push* adds an item whereas a *Pop* returns an item for a non-empty or $NULL$ for empty stack (line 27). Recall that the framework performs a special emptiness check before returning an empty *sub-stack*.

3.2 Optimizations

Our design framework can be tuned to optimize for; locality, contention and *hops* overhead, using the *width* and *depth* parameters.

3.2.1 Locality and Contention

To exploit locality, the thread starts its *Window* search from the previously known index on which it succeeded. This allows the thread a chance to operate on the same *sub-structure* multiple times locally, given that the *sub-structure* is valid. Working locally improves the caching behaviour, which in return improves performance especially under a NUMA execution environment with high communication cost across NUMA nodes [7, 16, 28].

A failed operation on a valid *sub-structure* signals the possibility of contention. The thread that fails on a `CAS` (Algorithm 2: line 11,24), starts the *Window* search on a randomly selected index (Algorithm 1: line 11). This reduces possible contention that might arise if the failed threads were to retry on the same *sub-structure*. Furthermore, random selection avoids contention on individual *sub-structures* by uniformly distributing the failed threads to all available *sub-structures*.

For every *Window* search, if the search start index is invalid, the thread tries a given number of random jumps (Algorithm 1: line 44), then switches to round robin (Algorithm 1: line 52) until a valid *sub-structure* is found. In our case, we use two random jumps as the optimal number for a random search basing on the power of random two choices [22]. However, this is a configurable parameter that can take any value.

We further note that contention is inversely proportional to the *width*. As a simple model, we split the latency of an operation into contention (op_{cont}) and contention-free (op_{free}) operation costs, given by $op = op_{cont}/width + op_{free}$. This means that we can increase the *width* to further reduce contention.

3.2.2 *hops* Overhead

The number of *hops* increases with an increase in *width*. This counteracts the performance benefits from contention reduction through increasing *width*, necessitating a balance between contention and *hops* reduction. Based on our simple contention model above, the performance would increase as the contention factor vanishes with the increase of *width*, but with an asymptote at $1/op_{free}$. This implies that beyond some point, one cannot really

gain throughput by increasing the *width*, however, throughput would get hurt due to the increased number of *hops*. At some point as *width* increases, gains from the contention factor ($\lim_{width\to\infty} op_{cont} \to 0$) are surpassed by the increasing cost of *hops*. To avoid this, we switch to increasing *depth* instead of *width*, at the point of *width* saturation. Increasing *depth* reduces the number of *hops*. This is supported by our step complexity analysis [27] given by Theorem 1. Where $p = P(Put)$ and $\mathbb{E}(Extra) = \mathbb{E}(hop) + \mathbb{E}(shift)$.

▶ **Theorem 1.** *For a 2Dc-structure that is initialized with parameters depth, width, $shift =$ depth and $p = 1/2$, $\mathbb{E}(Extra) = O(\frac{\ln width}{depth})$.*

4 Correctness

In this section, we prove the correctness of our 2D derived data structures, including their *relaxation* bounds and lock freedom. Due to space constraints, we present the *2Dc-Stack* correctness proofs and only give Theorem 2 for our other derived 2*D* data structures. We refer the reader to our extended version [27] for the rest of the proofs.

Each *sub-structure* is lock-free: An operation can fail on `CAS` only if there is another successful operation. For *WinCoupled*, *Window shifting* is lock-free iff $shift < depth$, whereas it is always lock-free for *WinDecoupled*. A *Window shift* can only fail if there is another successful *shift* operation preceded by a successful *Put* or *Get*, ensuring system progress. Thus, all our derived algorithms are lock-free. Our design framework can also be used for lock based data structures.

▶ **Theorem 2.** *All our derived 2D data structures are linearizable with respect to k-out-of-order semantics for the respective data structure Semitics. Where, for 2Dd-Stack $k = (3depth)(width{-}1)$, for 2Dd-Queue $k = (depth)(width{-}1)$, for 2Dd-Deque $k = (8depth)(width{-}1)$, for 2Dd-Counter and 2Dc-Counter $k = (2depth)(width - 1)$.*

2Dc-Stack is linearizable with respect to the sequential semantics of *k-out-of-order* stack [19]. *2Dc-Stack Push* and *Pop* linearization points are similar to those of the original Treiber's Stack. As shown in Algorithm 2, *Pop* linearizes either by returning *NULL* (line 27) or with a successful `CAS` (line 22). *Push* linearizes with a successful `CAS` (line 9).

Relaxation can be applied method-wise and it is applied only to *Pop* operations, that is, a *Pop* pops one of the topmost *k* items. Firstly, we require some notation. The *Window* defines the number of operations allowed to proceed on any given *sub-stack*. The *Window* is shifted by the parameter $shift$, $1 \le shift < depth$ and $width = \#sub\text{-}stacks$. For simplicity, let $shift = shift^{up} = shift_{down}$. A *Window i* (W_i) has an upper bound (W_i^{max}) and a lower bound (W_i^{min}), where $W_i^{max} = i \times shift$ and $W_i^{min} = (i \times shift) - depth$, respectively. For simplicity, let *Global* represent the current global upper bound (Win_{max}). A *Window* is active iff $W_i^{max} = Global$. The number of items of the *sub-stack j* is denoted by N_j, $1 \le j \le width$. To recall, the top pointer, the version number and N_j are embedded into the *descriptor* of *sub-stack j* and all can be modified atomically with a wide `CAS` instruction.

▶ **Lemma 3.** *Given that $Global = shift \times i$, it is impossible to observe a state(S) such that $N_j > W_{i+1}^{max}$ (or $N_j < W_{i-1}^{min}$).*

Proof. We show that this is impossible by considering the interleaving of operations. Without loss of generality, assume thread 1 (P_1) has set $Global = shift \times i$ at time $t_1\prime$. To do this, P_1 should have observed either $Global = shift \times (i-1)$ and then $N_j = W_{i-1}^{max}$ or $Global = shift \times (i+1)$ and then $N_j = W_{i+1}^{min}$. Let this observation of *Global* happen at time t_1. Consider the last successful push operation at *sub-stack j* before the state S is observed for the first time (we do not consider *Pop* operations as they can only decrease N_j to a value

that is less than W_{i+1}^{max}, this case will be covered by the first item below). Assume thread 0 (P_0) sets N_j to $N_j > W_{i+1}^{max}$ in this push operation. P_0 should observe $N_j \geq W_{i+1}^{max}$ and $Global > W_{i+1}^{max}$. Let j be selected at time t_0. And the linearization of the operation happens at $t_0\prime > t_0$.

- If $t_0\prime < t_1$, the concerned state(S) can not be observed since *Global* cannot be changed (to $shift \times i$) after $N_j > W_{i+1}^{max}$ is observed.
- Else if $t_1\prime < t_0$, the concerned state(S) cannot be observed since the push operation cannot proceed after observing *Global* with such N_j.
- Else if $t_1 > t_0$, then P_0 cannot linearize because, this implies N_j has been modified (the difference between the value of *Global* that is observed by P_0 and then by P_1 implies this) since P_0 had read the descriptor, the version numbers would have changed since then.
- Else if $t_1 < t_0$, then this implies *Global* has been modified, since it was read by P_1, thus updating *Global* would fail, at least based on the version number. ◀

▶ **Lemma 4.** *At all times, there exist an i such that $\forall j, 1 \leq j \leq width$: $W_i^{min} \leq N_j \leq W_{i+1}^{max}$.*

Proof. Informally, the lemma states that the size (number of operations) of a *sub-stack* spans to at most two consecutive accessible *windows*. Assume that the statement is not true, then there should exist a pair of *sub-stacks* (y and z) at some point in time such that $\exists i, N_y < W_i^{min}$ and $N_z > W_{i+1}^{max}$. Consider the last *Push* at *sub-stack* z and last *Pop* at *sub-stack* y that linearize before or at the time t.

Assume thread P_0 (*Push*) sets N_z and thread P_1 (*Pop*) sets N_y. To do this, P_0 should observe $N_z \geq W_{i+1}^{max}$ and $Global > W_{i+1}^{max}$, let *sub-stack* z be selected at t_0. And, the linearization of the *Push* operation occurs at $t_0\prime > t_0$. Similarly, for P_1 *Pop* operation, let *sub-stack* y be selected at t_1, P_1 should have observed $Global \leq W_i^{min}$. And, let the *Pop* operation linearize at time $t_1\prime > t_1$. Now, we consider the possible interleavings.

- If $t_0\prime < t_1$ (or the symmetric $t_1\prime < t_0$ for which we do not repeat the arguments), then for P_1 to proceed and pop an item from *sub-stack* y, it is required that $Global \leq W_i^{min}$. Based on Lemma 3, this is impossible when $N_z > W_i^{max}$.
- Else if $t_1 > t_0$, then P_0 cannot linearize, because this implies that N_z has been modified (the difference between the value of *Global* that is observed by P_0 and then by P_1 implies this) since P_0 has read the *descriptor*. The version number would have changed since then.
- Else if $t_0 > t_1$, the argument above holds for P_1 too, so P_1 should fail to linearize.

Such N_z and N_y pair can not co-exist at any time. ◀

▶ **Theorem 5.** *2Dc-Stack is linearizable with respect to k-out-of-order stack semantics, where $k = (2shift + depth)(width - 1)$.*

Proof. Consider the *Push* (t_e^{push}) and *Pop* (t_e^{pop}) linearization points, that insert and remove an item e for a given *sub-stack* j respectively, where, $t_e^{pop} > t_e^{push}$. Now, we bound the maximum number of items, that are pushed after t_e^{push} and are not popped before t_e^{pop}, to obtain k. Let item e be the ${N_j}^{th}$ item from the bottom of the *sub-stack*. Consider a *Window* i such that: $W_i^{min} \leq N_j \leq W_i^{max}$.

Lemma 4 states that the sizes of the *sub-stacks* should reside in a bounded region. Relying on Lemma 4, we can deduce that at time t_e^{push}, the following holds: $\forall i : N_j \geq W_i^{min} - shift$. Similarly, we can deduce that at time t_e^{pop}, the following holds: $\forall i : N_j \leq W_i^{max} + shift$. Therefore, the maximum number of items, that are pushed to *sub-stack* j after t_e^{push} and are not popped before t_e^{pop} is at most $W_i^{max} + shift - (W_i^{min} - shift) = depth + 2shift$. We know that this number is zero for *sub-stack* j (the *sub-stack* that e is inserted) and we have $width - 1$ other *sub-stacks*. So, there can be at most $(2shift + depth)(width - 1)$ items that are pushed after t_e^{push} and are not popped before t_e^{pop}. ◀

5 Experimental Evaluation

We experimentally evaluate the performance of our derived $2D$ algorithms, in comparison to *k-out-of-order relaxed* algorithms available in the literature, and other state of the art data structure algorithms. *k-out-of-order relaxed* algorithms include; Last recently used queue (*lru*) [15], Segmented queue (*Q-segment*) and *k-Stack* [1, 19], other algorithms include; MS-queue (*MS-queue*) [21], Wait free queue (*wfqueue*) [36], Time stamped stack (*TS-Stack*) [11] and Elimination back-off stack (*Elimination*) [18]. To facilitate a detailed study,we implement three extra *relaxation* techniques following the same multiple *sub-structures* design; *Random*, *Random-C2* and *Round-Robin*. These techniques present a combination of characteristics that add value to our evaluation. In order to compare to data structures that have used such techniques in the literature, we implement general data structures for each technique as we describe in the next paragraph. We shall use *width* generally to refer to #*sub-structures* for all algorithms using the multiple *sub-structures* design.

For *Random*, a *Put* or *Get* operation selects a *sub-structure* randomly and proceeds to operate on it, whereas for *Random-C2*, a *Get* operation randomly selects two *sub-structures*, compares their items returning the most correct depending on the data structure semantics [2, 3, 24, 25]. *Put* operations time stamp items marking their time of entry. It is these timestamps that are compared to determine the precedence order among the two items. Due to the randomized distribution of operations, we expect low or no contention, no locality, no *hops* and no deterministic *k-out-of-order* relaxation bound. We derive *S-random* and *S-random-c2* stacks, *Q-random* and *Q-random-c2* queues, *C-random* and *C-random-c2* counters for both *Random* and *Random-C2* respectively.

Under *Round-Robin*, a thread selects and operates once on a *sub-structure* in a strict round-robin order following its local counter. The thread must succeed on the selected *sub-structure* before proceeding to the next. Due to retries on the same *sub-structure*, we expect contention and no *hops*. The thread operates once on each *sub-structure*, hence no locality. We derive *S-robin* stack, *Q-robin* queue, and *C-robin* counter. *Round-Robin* provides *relaxation* bounds [27], we demonstrate this using *S-robin* bound given by Theorem 6.

▶ **Theorem 6.** *S-robin is linearizable with respect to k-out-of-order stack semantics, where* $k = (2 \times \#threads - 1)(\#sub\text{-}stacks - 1)$.

To facilitate a uniform comparison, we implemented all the evaluated algorithms using the same development tools. The source code will be made publicly available.

5.1 System Description

Experiments are run on two x86-64 machines: (i) Intel Xeon E5-2687W v2 machine with 2 sockets, 8-core Intel Xeon processors each running at 3.4GHz, L2 cache = 256KB, L3 cache = 25.6MB (*Multi-S*) and (ii) Intel Xeon Phi 7290 with one 72-core processor running at 1.5GHz, L2 cache = 1024KB (*Single-S*). *Multi-S* and *Single-S* run on Ubuntu 16.04.2 LTS and CentOS Linux 7 Core Operating systems receptively. The *Multi-S* machine is used to evaluate inter-socket execution behaviour, whereas *Single-S* is used to evaluate intra-socket. Threads are pined one per core, for both machines excluding hyper-threading. Inter socket execution is evaluated through pinning the threads one per socket in round robin fashion. Threads randomly select between *Put* or *Get* with a given probability (operation rate). Memory is managed using the ASCYLIB framework SSMEM [8].

Our main goal is to achieve scalability under high contention. To evaluate this, we simulate high contention by excluding work between operations. To reduce the effect of *Get* $NULL$ returns in our results, any given algorithm is initialized with 2^{17} items. Each experiment is then run for five seconds obtaining an average of five repeats. Throughput is

measured in terms of operations per second, whereas the *relaxation* behaviour (*accuracy*) is measured in terms of the error distance from the exact data structure sequential semantics [19]. The higher the distance, the lower the *accuracy*.

Our design framework is tunable, giving designers the ability to manage performance optimizations for different execution environments and workloads, within a given tight *relaxation* bound (k). This, however, calls for a multi-objective optimization model, which is beyond the scope of this paper. We instead run tuning experiments to obtain a tuned configuration for our evaluation. From our tuning experiments [27], we observe that $width = 3 \times \#threads$ provides a fair balance between *accuracy* and throughput for all $2D$ algorithms. We use this *width* configuration in the rest of the experiments.

5.2 Measuring *accuracy*

We adopt a similar methodology used in the literature [4, 24]. A sequential linked-list is run alongside the data structure being measured. For each operation *Put* or *Get*, a simultaneous insert or delete is performed on the linked-list respectively, following the exact semantics of the given data structure. A global lock is carefully placed at the data structure linearization points, locking both the linked-list and the data structure simultaneously. The lock allows only one thread to update both the data structure and the linked-list in isolation.

A given thread has to acquire the lock before it tries to linearize on any given *sub-structure*. Note that, *Window* search is independent of the lock. Items on the data structure are duplicated on the linked-list and can be identified by their unique labels. Insert operations happen at the head or tail of the list for LIFO or FIFO measurements respectively. A delete operation searches for the given item deletes it and returns its distance from the head (*error-distance*). For counter measurements, we replace the linked-list with a fetch and add (`FAA`) counter. Both counters are updated in isolation using a lock like explained above. The error distance is calculated from the difference between the two counter values.

Experiment results are then plotted using logarithmic scales, throughput (solid lines) and error distance (dotted lines) sharing the x-axis.

5.3 Monotonicity With High Degree of Relaxation

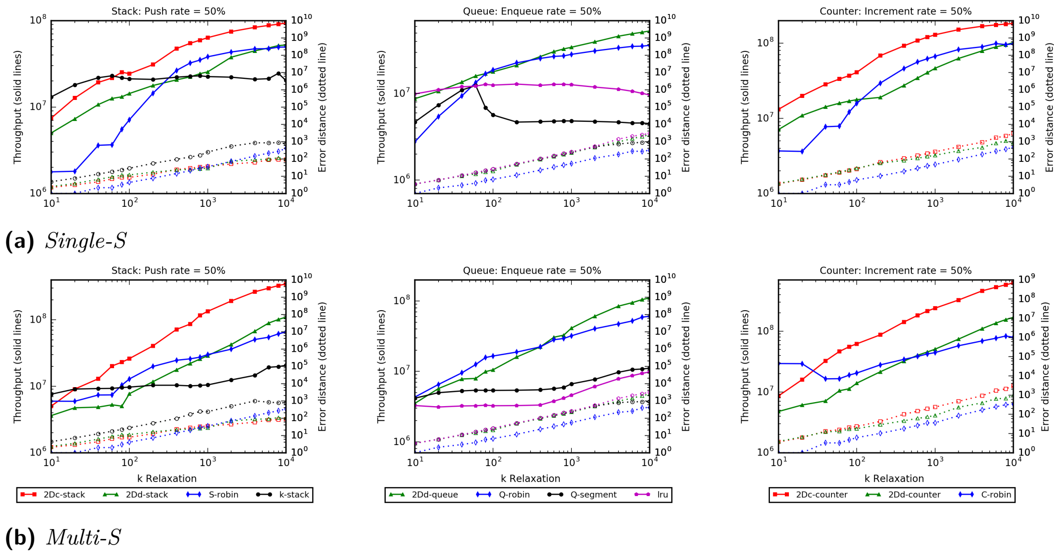
In order to evaluate monotonicity with increasing *relaxation* (k), we fix the number of threads to 16 as presented in Figure 2. This is to match the number of cores available on *Multi-S*.

First, we observe the difference between *WinCoupled* and *WinDecoupled* by comparing the *2Dc-Stack* and the *2Dd-Stack* respectively. *2Dc-Stack* consistently outperforms *2Dd-Stack* due to the reduced *Window shifting* updates. With *2Dc-Stack*, a given thread can locally operate on the same *sub-stack* longer since operation counts cancel out each other, leaving the *sub-stack* in a valid state. This increases the probability of exploiting locality.

All algorithms increase their *width* as k increases to reduce contention and allow for increased disjoint *access*. However, for *k-Stack*, *Q-segment*, and *lru*, *hops* increase as *width* increases, this explains their observed low throughput. *S-robin*, *Q-robin* and *C-robin* are not affected by *hops*. However, for smaller k values, they suffer from high contention arising from contending threads retrying on the same *sub-structure* until they succeed. As contention vanishes with high k values, throughput gain saturates due to lack of locality.

$2D$ algorithms maintain throughput gain through limiting *width* to a size beneficial to reducing contention and switch to adjusting the *depth* to reduce *hops*. The *depth* parameter allows $2D$ algorithms to maintain throughput gain through exploiting locality while reducing latency. This is observed for both *Single-S* and *Multi-S* machines.

In terms of *accuracy*, we observe an almost linear decrease in *accuracy* as k increases for all algorithms.



(a) *Single-S*

(b) *Multi-S*

Figure 2 Throughput and observed *accuracy* as k bound relaxation increases, with 16 threads.

5.4 Scaling With Threads

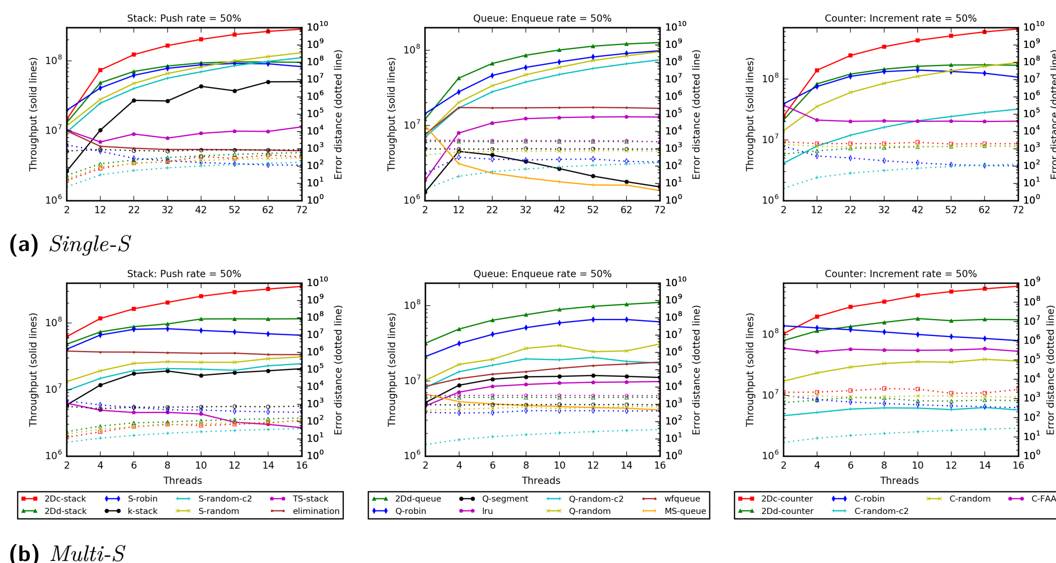
To evaluate scalability, we fix the *relaxation* bound to ($k = 10^4$) and vary the number of threads as shown in Figure 3. The reason for $k = 10^4$ is to reduce the effect of contention due to small *width* at lower k values. This helps us focus on scalability effects. *Random* and *Random-C2* algorithms' *width* is set to $3 \times \#threads$, as the optimal balance between throughput and *accuracy* since both of them do not provide k *relaxation* bounds as we mentioned before. *Random* and *Random-C2* *width* matches the 2*D* algorithms' *width* configuration, providing for a fair comparison.

For *Round-Robin* algorithms, the *width* is inversely proportional to #threads (see Theorem 6). As #threads increases, *width* reduces leading to increased contention. This explains the observed drop in throughput for a high number of threads, especially for the *S-robin* and the *C-robin* algorithms due to their *sub-structure* single *access*. The effect of lack of locality can be reduced by hardware pre-fetching, a feature available on both machines. This can also explain the *Round-Robin* better performance compared to the performance of the other algorithms that lack locality.

k-Stack and *Q-segment* maintain a constant segment size as the #threads increases. This increases the rate at which segments get filled up, leading to a high frequency of *hops* and segment maintenance cost. As observed, throughput gain saturates at high #threads leading to limited scalability.

The scalability of *lru* is limited by the global counter used to calculate the last recently used *sub-queue*. For every operation, the thread has to increment the global counter using a `FAA` instruction, turning the counter into a scalability bottleneck. This can be observed when *lru* performance is compared against that of a single `FAA` counter (*C-FAA*). *wfqueue* suffers from the same `FAA` counter sequential bottleneck.

Random and *Random-C2* algorithms are affected by the lack of locality, which is evident by the difference between *Single-S* and *Multi-S* results. We observe that the performance difference between *Random* and 2*D* algorithms increases on the *Multi-S* (Figure 3b) machine as compared to that on the *Single-S* (Figure 3b) machine. This demonstrates how much 2*D* algorithms gain from exploiting locality when executing on a *Multi-S* machine. Locality helps to avoid paying the high inter-socket communication cost [7, 16, 28].



(a) *Single-S*

(b) *Multi-S*

Figure 3 Throughput and observed *accuracy* as the number of threads increases, with $k = 10^4$.

TS-Stack's throughput is limited by the *Pop* search retries, searching for the newest item. Moreover, *Pop* operations might contend on the same newest items if there are not enough concurrent *Push* operations. Also, *Pop* lacks locality, which explains the drop in throughput on the *Multi-S* machine, due to the high inter-socket communication costs.

We observe an increase in the *accuracy* loss as the number of threads increase for $2D$ algorithms, *Random* and *Random-C2*. This is due to the increase in *width* as threads increase in number. *Round-Robin* algorithms show an increase in *accuracy* due to their decrease in *width*, whereas *lru*, *Q-segment* and *k-Stack* show little change in *accuracy* since the *width* and segment size does not change for the different number of threads respectively.

6 Conclusion

In this work, we have shown that semantics *relaxation* has the potential to monotonically trade relaxed semantics of concurrent data structures for achieving throughput performance within tight *relaxation* bounds. This has been achieved through an efficient two-dimensional framework that is simple and easy to implement for different data structures. We demonstrated that, by deriving two-dimensional lock-free designs for stacks, FIFO queues, dequeues and shared counters.

Our experimental results have shown that *relaxing* in one dimension, restricts the capability to control *relaxation* behaviour in-terms of throughput and *accuracy*. Compared to previous solutions, our framework can be used to extend existing data structures with minimal modifications while achieving better performance in terms of throughput and *accuracy*.

References

1 Yehuda Afek, Guy Korland, and Eitan Yanovsky. Quasi-linearizability: Relaxed consistency for improved concurrency. In *International Conference on Principles of Distributed Systems*, pages 395–410. Springer, 2010.

2 Dan Alistarh, Trevor Brown, Justin Kopinsky, Jerry Zheng Li, and Giorgi Nadiradze. Distributionally Linearizable Data Structures. *CoRR*, abs/1804.01018, 2018. `arXiv:1804.01018`.

3 Dan Alistarh, Justin Kopinsky, Jerry Li, and Giorgi Nadiradze. The Power of Choice in Priority Scheduling. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC '17, pages 283–292, 2017.

4 Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. The SprayList: A scalable relaxed priority queue. *ACM SIGPLAN Notices*, 50(8):11–20, 2015.

5 Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin Vechev. Laws of Order: Expensive Synchronization in Concurrent Algorithms Cannot Be Eliminated. *SIGPLAN Not.*, 46(1):487–498, January 2011.

6 Gal Bar-Nissan, Danny Hendler, and Adi Suissa. A Dynamic Elimination-Combining Stack Algorithm. *CoRR*, abs/1106.6304, 2011. arXiv:1106.6304.

7 Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 33–48, 2013.

8 Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. *SIGARCH Comput. Archit. News*, 43(1):631–644, March 2015.

9 Edsger W. Dijkstra. The Structure of "THE"-multiprogramming System. *Commun. ACM*, 11(5):341–346, May 1968.

10 EW Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

11 Mike Dodds, Andreas Haas, and Christoph M. Kirsch. A Scalable, Correct Time-Stamped Stack. *SIGPLAN Not.*, 50(1):233–246, January 2015.

12 Panagiota Fatourou and Nikolaos D. Kallimanis. Revisiting the Combining Synchronization Technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 257–266, 2012.

13 Elad Gidron, Idit Keidar, Dmitri Perelman, and Yonathan Perez. SALSA: scalable and low synchronization NUMA-aware algorithm for producer-consumer pools. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, pages 151–160. ACM, 2012.

14 Andreas Haas, Thomas A. Henzinger, Andreas Holzer, Christoph M. Kirsch, Michael Lippautz, Hannes Payer, Ali Sezgin, Ana Sokolova, and Helmut Veith. Local Linearizability for Concurrent Container-Type Data Structures. In *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada*, pages 6:1–6:15, 2016. doi:10.4230/LIPIcs.CONCUR.2016.6.

15 Andreas Haas, Michael Lippautz, Thomas A. Henzinger, Hannes Payer, Ana Sokolova, Christoph M. Kirsch, and Ali Sezgin. Distributed Queues in Shared Memory: Multicore Performance and Scalability Through Quantitative Relaxation. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '13, pages 17:1–17:9, 2013.

16 Daniel Hackenberg, Daniel Molka, and Wolfgang E. Nagel. Comparing Cache Architectures and Coherency Protocols on x86-64 Multicore SMP Systems. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 413–422, 2009.

17 Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat Combining and the Synchronization-parallelism Tradeoff. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 355–364, 2010.

18 Danny Hendler, Nir Shavit, and Lena Yerushalmi. A Scalable Lock-free Stack Algorithm. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, pages 206–215, 2004.

19 Thomas A. Henzinger, Christoph M. Kirsch, Hannes Payer, Ali Sezgin, and Ana Sokolova. Quantitative Relaxation of Concurrent Data Structures. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 317–328, 2013.

20 Maged M. Michael. CAS-Based Lock-Free Algorithm for Shared Deques. In *Euro-Par 2003. Parallel Processing, 9th International Euro-Par Conference, Klagenfurt, Austria, August 26-29, 2003. Proceedings*, pages 651–660, 2003. doi:10.1007/978-3-540-45209-6_92.

21 Maged M. Michael and Michael L. Scott. Simple, Fast, and Practical Non-blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, pages 267–275, 1996.

22 Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

23 Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit. Using Elimination to Implement Scalable and Lock-free FIFO Queues. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '05, pages 253–262, 2005.

24 Hamza Rihani, Peter Sanders, and Roman Dementiev. Brief announcement: Multiqueues: Simple relaxed concurrent priority queues. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*, pages 80–82. ACM, 2015.

25 Adones Rukundo, Aras Atalar, and Philippas Tsigas. 2D-Stack: A scalable lock-free stack design that continuously relaxes semantics for better performance. Technical report, Chalmers University of Technology, 2018. URL: https://research.chalmers.se/publication/506089/file/506089_Fulltext.pdf.

26 Adones Rukundo, Aras Atalar, and Philippas Tsigas. Brief Announcement: 2D-Stack - A Scalable Lock-Free Stack Design that Continuously Relaxes Semantics for Better Performance. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018, Egham, United Kingdom, July 23-27, 2018*, pages 407–409, 2018. URL: https://dl.acm.org/citation.cfm?id=3212794.

27 Adones Rukundo, Aras Atalar, and Philippas Tsigas. Monotonically relaxing concurrent data-structure semantics for performance: An efficient 2D design framework. *CoRR*, abs/1906.07105, 2019. arXiv:1906.07105.

28 Hermann Schweizer, Maciej Besta, and Torsten Hoefler. Evaluating the Cost of Atomic Operations on Modern Architectures. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*, PACT '15, pages 445–456, 2015.

29 Nir Shavit. Data Structures in the Multicore Age. *Commun. ACM*, 54(3):76–84, March 2011.

30 Nir Shavit and Gadi Taubenfeld. The Computability of Relaxed Data Structures: Queues and Stacks As Examples. *Distrib. Comput.*, 29(5):395–407, October 2016.

31 Nir Shavit and Dan Touitou. Elimination Trees and the Construction of Pools and Stacks: Preliminary Version. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '95, pages 54–63, 1995.

32 Nir Shavit and Asaph Zemach. Combining funnels: a dynamic approach to software combining. *Journal of Parallel and Distributed Computing*, 60(11):1355–1387, 2000.

33 Edward Talmage and Jennifer L. Welch. *Relaxed Data Types as Consistency Conditions*, pages 142–156. Springer International Publishing, Cham, 2017.

34 R.K. Treiber. *Systems Programming: Coping with Parallelism.* Research Report RJ. International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

35 Martin Wimmer, Jakob Gruber, Jesper Larsson Träff, and Philippas Tsigas. The Lock-free k-LSM Relaxed Priority Queue. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 277–278, 2015.

36 Chaoran Yang and John Mellor-Crummey. A Wait-free Queue As Fast As Fetch-and-add. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 16:1–16:13, 2016.

Phase Transitions of Best-of-Two and Best-of-Three on Stochastic Block Models

Nobutaka Shimizu
The University of Tokyo, Japan
nobutaka_shimizu@mist.i.u-tokyo.ac.jp

Takeharu Shiraga
Chuo University, Tokyo, Japan
shiraga.076@g.chuo-u.ac.jp

Abstract

This paper is concerned with voting processes on graphs where each vertex holds one of two different opinions. In particular, we study the *Best-of-two* and the *Best-of-three*. Here at each synchronous and discrete time step, each vertex updates its opinion to match the majority among the opinions of two random neighbors and itself (the Best-of-two) or the opinions of three random neighbors (the Best-of-three). Previous studies have explored these processes on complete graphs and expander graphs, but we understand significantly less about their properties on graphs with more complicated structures.

In this paper, we study the Best-of-two and the Best-of-three on the stochastic block model $G(2n, p, q)$, which is a random graph consisting of two distinct Erdős-Rényi graphs $G(n, p)$ joined by random edges with density $q \leq p$. We obtain two main results. First, if $p = \omega(\log n/n)$ and $r = q/p$ is a constant, we show that there is a phase transition in r with threshold r^* (specifically, $r^* = \sqrt{5} - 2$ for the Best-of-two, and $r^* = 1/7$ for the Best-of-three). If $r > r^*$, the process reaches consensus within $O(\log\log n + \log n/\log(np))$ steps for any initial opinion configuration with a bias of $\Omega(n)$. By contrast, if $r < r^*$, then there exists an initial opinion configuration with a bias of $\Omega(n)$ from which the process requires at least $2^{\Omega(n)}$ steps to reach consensus. Second, if p is a constant and $r > r^*$, we show that, for any initial opinion configuration, the process reaches consensus within $O(\log n)$ steps. To the best of our knowledge, this is the first result concerning multiple-choice voting for arbitrary initial opinion configurations on non-complete graphs.

2012 ACM Subject Classification Mathematics of computing → Stochastic processes; Mathematics of computing → Random graphs

Keywords and phrases Distributed Voting, Consensus Problem, Random Graph

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.32

Related Version The full-version is [36], `https://arxiv.org/abs/1907.12212`.

Funding *Nobutaka Shimizu*: JST CREST Grant Number JPMJCR14D2 and JSPS KAKENHI Grant Number 19J12876, Japan
Takeharu Shiraga: JSPS KAKENHI Grant Number 17H07116 and 19K20214, Japan

Acknowledgements We would like to thank Colin Cooper, Nan Kang and Tomasz Radzik for helpful discussions. We also thank the anonymous reviewers for their helpful comments.

1 Introduction

This paper is concerned with *voting processes* on distributed networks. Consider an undirected connected graph $G = (V, E)$ where each vertex $v \in V$ initially holds an opinion from a finite set. A voting process is defined by a local updating rule: Each vertex updates its opinion according to the rule. Voting processes appear as simple mathematical models in a wide range of fields, e.g. social behavior, physical phenomena and biological systems [32, 30, 4]. In distributed computing, voting processes are known as a simple approach for consensus problems [20, 23].

© Nobutaka Shimizu and Takeharu Shiraga;
licensed under Creative Commons License CC-BY
33rd International Symposium on Distributed Computing (DISC 2019).
Editor: Jukka Suomela; Article No. 32; pp. 32:1–32:17

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1.1 Previous work

The synchronous *pull voting* (a.k.a. the *voter model*) is a simple and well-studied voting process [33, 25]. In the pull voting, at each synchronous and discrete time step, each vertex adopts the opinion of a randomly selected neighbor. Here, the main quantity of interest is the *consensus time*, which is the number of steps required to reach consensus (i.e. the configuration where all vertices hold the same opinion). Hassin and Peleg [25] showed that the expected consensus time is $O(n^3 \log n)$ for all non-bipartite graphs and for all initial opinion configurations, where n is the number of vertices. Note that, for bipartite graphs, there exists an initial opinion configuration that never reaches consensus.

The pull voting has been extended to develop voting processes where each vertex queries multiple neighbors at each step. The simplest multiple-choice voting process is the *Best-of-two* (*two sample voting*, or *2-Choices*), where each vertex $v \in V$ randomly samples two neighbors (with replacement) and, if both hold the same opinion, adopts it[1]. Doerr et al. [19] showed that, for complete graphs initially involving two possible opinions, the consensus time of the Best-of-two is $O(\log n)$ with high probability[2]. Likewise, the *Best-of-three* (a.k.a. *3-Majority*) is another simple multiple-choice voting process where each vertex adopts the majority opinion among those of three randomly selected neighbors. Several researchers have studied this model on complete graphs initially involving $k \geq 2$ opinions [8, 7, 10, 22]. For example, Ghaffari and Lengler [22] showed that the consensus time of the Best-of-three is $O(k \log n)$ if $k = O(n^{1/3}/\sqrt{\log n})$.

Several studies of multiple-choice voting processes on non-complete graphs have considered expander graphs with an *initial bias*, i.e. a difference between the initial sizes of the largest and the second largest opinions. Cooper et al. [13] showed that, for any regular expander graph initially involving two opinions, the Best-of-two reaches consensus within $O(\log n)$ steps w.h.p. if the initial bias is $\Omega(n\lambda_2)$, where λ_2 is the second largest eigenvalue of the graph's transition matrix. This result was later extended to general expander graphs, including Erdős-Rényi random graphs $G(n,p)$, under milder assumptions about the initial bias [14]. Recall that the Erdős-Rénri graph $G(n,p)$ is a graph on n vertices where each vertex pair is joined by an edge with probability p, independent of any other pairs. In [15], the authors studied the Best-of-two and the Best-of-three on regular expander graphs initially involving more than two opinions. In [3, 28], the authors studied multiple-choice voting processes on non-complete graphs with random initial configuration.

Recently, the Best-of-two on richer classes of graphs involving two opinions have been studied. Previous works proved interesting results which do not hold on complete graphs or expander graphs. Cruciani et al. [17] studied the Best-of-two on the *core periphery network*, namely a graph consisting of core vertices and periphery vertices. They showed that a phase transition can occur, depending on the density of edges between core and periphery vertices: Either the process reaches consensus within $O(\log n)$ steps, or remains a configuration where both opinions coexist for at least $\Omega(n)$ steps. Cruciani et al. [18] studied the Best-of-two on the (a, b)-*regular stochastic block model*, which is a graph consisting of two a-regular graphs connected by a b-regular bipartite graph. Under certain assumptions including $b/a = O(n^{-0.5})$, they showed that, starting from a random initial opinion configuration, the process reaches an almost *clustered* configuration (e.g. both communities are in almost consensus but the opinions are distinct) within $O(\log n)$ steps with constant probability, then stays in that configuration for at least $\Omega(n)$ steps w.h.p. They also proposed a distributed community detection algorithm based on this property.

[1] If the graph initially involves two possible opinions, this definition matches the rule described in Abstract.
[2] In this paper "with high probability" (w.h.p.) means probability at least $1 - n^{-c}$ for a constant $c > 0$.

1.2 Our results

This paper considers the *stochastic block model*, a well-known random graph model that forms multiple communities. This model has been well-explored in a wide range of fields, including biology [11, 31], network analysis [5, 24] and machine learning [2, 1], where it serves as a benchmark for community detection algorithms. The study of the voting processes on the stochastic block model has a potential application in distributed community detection algorithms [6, 9, 18]. In this paper, we focus on the following model which admits two communities of equal size.

▶ **Definition 1** (Stochastic block model). *For $n \in \mathbb{N}$ and $p, q \in [0,1]$ with $q \le p$, the stochastic block model $G(2n,p,q)$ is a graph on a vertex set $V = V_1 \cup V_2$, where $|V_1| = |V_2| = n$ and $V_1 \cap V_2 = \emptyset$. In addition, each pair $\{u,v\}$ of distinct vertices $u \in V_i$ and $v \in V_j$ forms an edge with probability θ, independent of any other edges, where*

$$\theta = \begin{cases} p & \text{if } i = j, \\ q & \text{otherwise.} \end{cases}$$

Note that $G(2n,p,q)$ is not connected w.h.p. if $p = o(\log n/n)$ [21]. Throughout this paper, we assume $p = \omega(\log n/n)$, in which regime each community is connected w.h.p.

In this paper, we first generate a random graph $G(2n,p,q)$, and then set an initial opinion configuration from $\{1,2\}$. Let $A^{(0)}, A^{(1)}, \dots$ be a sequence of random vertex subsets where $A^{(t)}$ is the set of vertices of opinion 1 at step t. For any $A \subseteq V$, the consensus time $T_{\mathrm{cons}}(A)$ is defined as

$$T_{\mathrm{cons}}(A) := \min\left\{ t \ge 0 \,:\, A^{(t)} \in \{\emptyset, V\},\, A^{(0)} = A \right\}.$$

We obtain two main results, described below.

Result I: phase transition

Observe that, if $p = q = 1$, then $G(2n,1,1)$ is a complete graph and the consensus time of the Best-of-two is $O(\log n)$, from the results of [19]. On the other hand, the graph $G(2n,1,0)$ consists of two disjoint complete graphs, each of size n, meaning that, depending on the initial state, it may not reach consensus. This naturally raises the following question: *Where is the boundary between these two phenomena?* This motivated us to study the consensus times of the Best-of-two and the Best-of-three on $G(2n,p,q)$ for a wide range of $r := q/p$, and led us to propose the following answers.

▶ **Theorem 2** (Phase transition of the Best-of-three on $G(2n,p,q)$). *Consider the Best-of-three on $G(2n,p,q)$ such that $r := \frac{q}{p}$ is a constant.*

(i) *If $r > \frac{1}{7}$, then $G(2n,p,q)$ w.h.p. satisfies the following property: There exist two positive constants $C, C' > 0$ such that*

$$\forall A \subseteq V \text{ of } \big||A| - |V \setminus A|\big| = \Omega(n): \\ \mathbf{Pr}\left[T_{\mathrm{cons}}(A) \le C\left(\log\log n + \frac{\log n}{\log(np)}\right)\right] \ge 1 - O\left(n^{-C'}\right).$$

(ii) *If $r < \frac{1}{7}$, then $G(2n,p,q)$ w.h.p. satisfies the following property: There exist a set $A \subseteq V$ with $\big||A| - |V \setminus A|\big| = \Omega(n)$ and two positive constants $C, C' > 0$ such that*

$$\mathbf{Pr}\left[T_{\mathrm{cons}}(A) \ge \exp(Cn)\right] \ge 1 - O\left(n^{-C'}\right).$$

▶ **Theorem 3** (Phase transition of the Best-of-two on $G(2n,p,q)$). *Consider the Best-of-two on $G(2n,p,q)$ such that $r := \frac{q}{p}$ is a constant.*

(i) *If $r > \sqrt{5}-2$, then $G(2n,p,q)$ w.h.p. satisfies the following property: There exist two positive constants $C, C' > 0$ such that*

$$\forall A \subseteq V \text{ of } \big||A| - |V \setminus A|\big| = \Omega(n):$$
$$\mathbf{Pr}\left[T_{\mathrm{cons}}(A) \le C\left(\log\log n + \frac{\log n}{\log(np)}\right)\right] \ge 1 - O\left(n^{-C'}\right).$$

(ii) *If $r < \sqrt{5}-2$, then $G(2n,p,q)$ w.h.p. satisfies the following property: There exist a set $A \subseteq V$ with $\big||A| - |V \setminus A|\big| = \Omega(n)$ and two positive constants $C, C' > 0$ such that*

$$\mathbf{Pr}\left[T_{\mathrm{cons}}(A) \ge \exp(Cn)\right] \ge 1 - O\left(n^{-C'}\right).$$

Note that the upper bound $T_{\mathrm{cons}}(A) = O(\log\log n + \log n/\log(np))$ is tight up to a constant factor if $\log n/\log(np) \ge \log\log n$. To see this, observe that there exists an $A \subseteq V$ such that $T_{\mathrm{cons}}(A)$ is at least half of the diameter. In addition, it is easy to see that the diameter of $G(2n,p,q)$ is $\Theta(\log n/\log(np))$ w.h.p. [21].

We also note that the consensus time of the pull voting is $O(\mathrm{poly}(n))$ for any non-bipartite graph [25]. To the best of our knowledge, Theorem 2 and Theorem 3 provide the first nontrivial graphs where the consensus time of a multiple-choice voting process is exponentially slower than that of the pull voting.

Result II: worst-case analysis

The most central topic in voter processes is the *symmetry breaking*, i.e. the number of iterations required to cause a small bias starting from the half-and-half state. Here, we are interested in the worst-case consensus time with respect to initial opinion configurations. To the best of our knowledge, all current results on worst-case consensus time of multiple-choice voting processes deal with complete graphs [19, 7, 10, 22]. All previous work on non-complete graphs has involved some special bias setting (e.g. an initial bias [13, 14, 15], or a random initial opinion configuration [3, 18, 28]). In this paper, we present the following first worst-case analysis of non-complete graphs.

▶ **Theorem 4** (Worst-case analysis of the Best-of-three on $G(2n,p,q)$). *Consider the Best-of-three on $G(2n,p,q)$ such that p and q are positive constants. If $\frac{q}{p} > \frac{1}{7}$, then $G(2n,p,q)$ w.h.p. satisfies the following property: There exist two positive constants $C, C' > 0$ such that*

$$\forall A \subseteq V: \mathbf{Pr}\left[T_{\mathrm{cons}}(A) \le C\log n\right] \ge 1 - O\left(n^{-C'}\right).$$

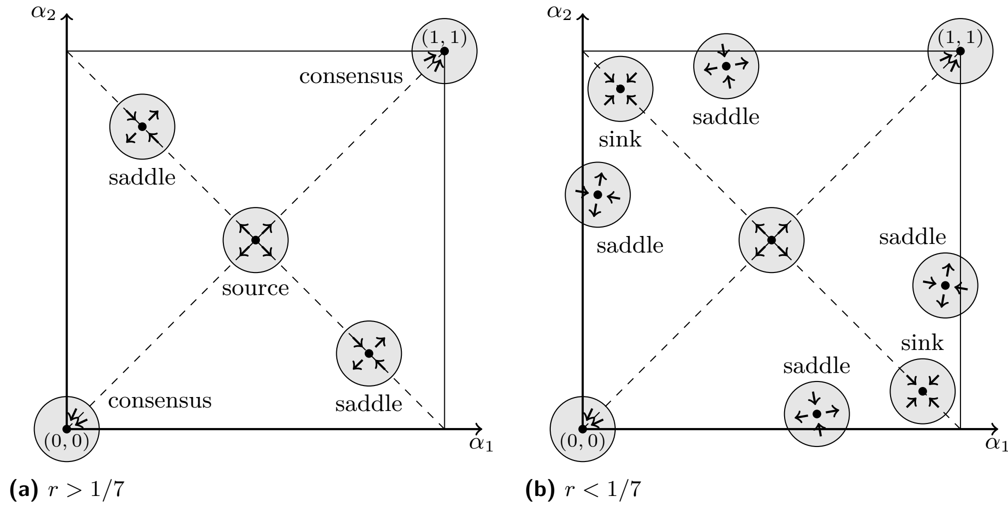
▶ **Theorem 5** (Worst-case analysis of the Best-of-two on $G(2n,p,q)$). *Consider the Best-of-two on $G(2n,p,q)$ such that p and q are positive constants. If $\frac{q}{p} > \sqrt{5}-2$, then $G(2n,p,q)$ w.h.p. satisfies the following property: There exist two positive constants $C, C' > 0$ such that*

$$\forall A \subseteq V: \mathbf{Pr}\left[T_{\mathrm{cons}}(A) \le C\log n\right] \ge 1 - O\left(n^{-C'}\right).$$

Based on these theorems, an immediate but important corollary follows.

▶ **Corollary 6.** *For any constant $p > 0$, the Best-of-two and the Best-of-three on the Erdős-Rényi graph $G(n,p)$ reach consensus within $O(\log n)$ steps w.h.p. for all initial opinion configurations.*

Recall that the Best-of-two and the Best-of-three on $G(n,p)$ has been extensively studied in previous works but these works put aforementioned assumptions on initial bias.



(a) $r > 1/7$ **(b)** $r < 1/7$

Figure 1 Four types of zero areas are illustrated. The sink areas do not appear if $r > 1/7$.

1.3 Strategy

Known techniques and our technical contribution

Consider a voting process on a graph $G = (V, E)$ where each vertex holds an opinion from $\{1, 2\}$, and let A be the set of vertices holding opinion 1. In general, a voting process with two opinions can be seen as a Markov chain with the state space $\{1, 2\}^V$. For $A \subseteq V$, let A' denote the set of vertices that hold opinion 1 in the next time step. Then, $|A'| = \sum_{v \in V} \mathbb{1}_{v \in A'}$ is the sum of independent random variables; thus, $|A'|$ concentrates on $\mathbf{E}[|A'| \mid A]$.

If the underlying graph is a complete graph, the state space can be regarded as $\{0, \dots, n\}$ (each state represents $|A|$). Therefore, $\mathbf{E}[|A'| \mid A]$ is expressed as a function of $|A|$, e.g. in the Best-of-two, $\mathbf{E}[|A'| \mid A] = f(|A|) := |A|\big(1 - (\frac{|A|}{n})^2\big) + (n - |A|)\big(\frac{|A|}{n}\big)^2 = n\big(3\big(\frac{|A|}{n}\big)^2 - 2\big(\frac{|A|}{n}\big)^3\big)$. Doerr et al. [19] exploited this idea for the Best-of-two and obtained the worst-case analysis for the consensus time on complete graphs. Somewhat interestingly, we also have $\mathbf{E}[|A'| \mid A] = f(|A|)$ in the Best-of-three.

Cooper et al. [13] extended this approach to the Best-of-two on regular expander graphs. Specifically, they proved that $\mathbf{E}[|A'| \mid A] = f(|A|) \pm O(\epsilon)$ for all $A \subseteq V$, where $\epsilon = \epsilon(n, \lambda_2) = o(n)$ is some function using the *expander mixing lemma*. This argument assumes an initial bias of size $\Omega(\epsilon)$. In another paper, Cooper et al. [14] improved this technique and proved more sophisticated results that hold for general (i.e. not necessarily regular) expander graphs.

In this paper, we consider $G(2n, p, q)$ on the vertex set $V = V_1 \cup V_2$. Let $A_i := A \cap V_i$ for $A \subseteq V$ and $i = 1, 2$. We prove that $G(2n, p, q)$ w.h.p. satisfies $\mathbf{E}[|A'_i| \mid A] = F_i(|A_1|, |A_2|) \pm O(\sqrt{n/p})$ for all $A \subseteq V$ in the Best-of-three, where $F_i : \mathbb{N}^2 \to \mathbb{N}$ is some function $(i = 1, 2)$. See (2) for details. We show the same result for the Best-of-two. Here, our key tool is the concentration method, specifically the Janson inequality [21] and the Kim-Vu concentration [29].

High-level proof sketch

Consider the Best-of-three on $G(2n, p, q)$, and let $A^{(0)}, A^{(1)}, \dots$ be a sequence of random vertex subsets determined by $A^{(t+1)} := (A^{(t)})'$ for each $t \geq 0$. Consider a stochastic process $\boldsymbol{\alpha}^{(t)} = (\alpha_1^{(t)}, \alpha_2^{(t)}) \in [0, 1]^2$ where $\alpha_i^{(t)} = |A^{(t)} \cap V_i|/n$ for $i = 1, 2$. Our technical result in the

previous paragraph approximates the stochastic process $\boldsymbol{\alpha}^{(t)}$ by the *deterministic* process $\mathbf{a}^{(t)}$ defined as $\mathbf{a}^{(t+1)} = H(\mathbf{a}^{(t)})$ and $\boldsymbol{\alpha}^{(0)} = \mathbf{a}^{(0)}$ for some function $H : [0,1]^2 \to [0,1]^2$ (See (4) and Figure 2). The function H induces a two-dimensional dynamical system, which we call the *induced dynamical system*. Using this, we obtain two results concerning $\boldsymbol{\alpha}^{(t)}$.

First, we show that, for any initial configuration, the process reaches one of the *zero areas* (a neighbor of a fixed point of H) within a constant number of steps. To show this, in addition to the approximation result, we used the theory of *competitive dynamical systems* [26].

Second, we characterize the behavior of $\boldsymbol{\alpha}^{(t)}$ in zero areas. The zero areas depend only on $r = q/p$, and are classified into four types using the Jacobian matrix: consensus, sink, saddle and source areas (see Figure 1 for a description). In consensus areas, we show that the process reaches consensus within $O(\log\log n + \log n/\log(np))$ steps. In sink areas, we show that the process remains there for at least $2^{\Omega(n)}$ steps, and also that sink areas only appear if $r < 1/7$. In saddle and source areas, we show that the process escapes from there within $O(\log n)$ steps if p is a constant by using techniques of [19]. Intuitively speaking, in these two kinds of areas, there are drifts towards outside. To apply the techniques of [19], we show that $\mathbf{Var}[|A'_i|] = \Omega(n)$ in the area if p is constant, which leads to our worst-case analysis result. Indeed, any previous works working on expander graphs did not investigate the worst-case due to the lack of variance estimation.

These arguments also enable us to study the Best-of-two process, which implies Theorem 3.

1.4 Related work

The consensus time of the pull voting process is investigated via its dual process, known as *coalescing random walks* [25, 12, 16]. Recently coalescing random walks have been extensively studied, including the relationship with properties of random walks such as the hitting time and the mixing time [27, 34].

Other studies have focused on voting processes with more general updating rules. Cooper and Rivera [16] studied the *linear voting model*, whose updating rule is characterized by a set of $n \times n$ binary matrices. This model covers the synchronous pull and the asynchronous push/pull voting processes. However, it does not cover the Best-of-two and the Best-of-three. Schoenebeck and Yu [35] studied asynchronous voting processes whose updating functions are *majority-like* (including the asynchronous *Best-of-*$(2k+1)$ voting processes). They gave upper bounds on the consensus times of such models on dense Erdős-Rényi random graphs using a potential technique.

Organization

First we set notation and precise definition of the Best-of-three in Section 2. After explaining key properties of the stochastic block model in Section 3, we show some auxiliary results of the induced dynamical system in Section 4. Then we derive Theorems 2 and 4 in Section 5. Our general framework of voting processes and results of the general induced dynamical systems are given in Section 6 and Section 7, respectively. Due to the page limitation, we omit detailed proofs and the discussion of the Best-of-two. See the full paper [36] for details.

2 Best-of-three voting process

For an $\ell \in \mathbb{N}$, let $[\ell] := \{1, 2, \ldots, \ell\}$. For a graph $G = (V, E)$ and $v \in V$, let $N(v)$ be the set of vertices adjacent to v. Denote the degree of $v \in V$ by $\deg(v) = |N(v)|$. For $v \in V$ and $S \subseteq V$, let $\deg_S(v) = |S \cap N(v)|$. Here, we study the Best-of-three with two possible opinions from $\{1, 2\}$.

▶ **Definition 7** (Best-of-three). *Let $G = (V, E)$ be a graph where each vertex holds an opinion from $\{1, 2\}$. Let*

$$f^{\mathrm{Bo3}}(x) := \binom{3}{3}x^3 + \binom{3}{2}x^2(1-x) = 3x^2 - 2x^3.$$

For the set A of vertices holding opinion 1, let A' denote the set of vertices that hold opinion 1 after an update. In the Best-of-three, $A' = \{v \in V : X_v = 1\}$ where $(X_v)_{v\in V}$ are independent binary random variables satisfying

$$\mathbf{Pr}[X_v = 1] = f^{\mathrm{Bo3}}\left(\frac{\deg_A(v)}{\deg(v)}\right).$$

For a given vertex subset $A^{(0)} \subseteq V$, we are interested in the behavior of the Markov chain $(A^{(t)})_{t=0}^{\infty}$, i.e. the sequence of random vertex subsets determined by $A^{(t+1)} := (A^{(t)})'$ for each $t \geq 0$. Let $A_i := V_i \cap A$ for $A \subseteq V$ and $i = 1, 2$. Since $|A'_i| = \sum_{v\in V_i} X_v$, the Hoeffding bound implies that the following holds w.h.p for $i = 1, 2$:

$$||A'_i| - \mathbf{E}[|A'_i|]| = O(\sqrt{n \log n}). \tag{1}$$

3 Concentration result for the stochastic block model

In this paper, we consider the Best-of-three on the stochastic block model $G(2n, p, q)$ (Definition 1). Then, $\mathbf{E}[|A'_i|]$ in (1) is a random variable since $G(2n, p, q)$ is a random graph. Here, our key ingredient is the following general concentration result for $G(2n, p, q)$.

▶ **Definition 8** (f-good $G(2n, p, q)$). *For a given function $f : [0, 1] \to [0, 1]$, we say $G(2n, p, q)$ is f-good if $G(2n, p, q)$ satisfies the following properties.*

(P1) *It is connected and non-bipartite.*

(P2) *A positive constant C_1 exists such that, for all $A, S \subseteq V$ and $i \in \{1, 2\}$,*

$$\left|\sum_{v\in S\cap V_i} f\left(\frac{\deg_A(v)}{\deg(v)}\right) - |S \cap V_i| f\left(\frac{|A_i|p + |A_{3-i}|q}{n(p+q)}\right)\right| \leq C_1\sqrt{\frac{n}{p}}.$$

(P3) *A positive constant C_2 exists such that, for all $A \subseteq V$, $S \in \{A, V \setminus A, V\}$ and $i \in \{1, 2\}$,*

$$\sum_{v\in S\cap V_i} f\left(\frac{\deg_A(v)}{\deg(v)}\right) \leq |S \cap V_i| f\left(\frac{|A_i|p + |A_{3-i}|q}{n(p+q)}\right) + C_2|A|\sqrt{\frac{\log n}{np}}.$$

▶ **Theorem 9** (Main technical theorem). *Suppose that $f : [0, 1] \to [0, 1]$ is a polynomial function with constant degree, $p = \omega(\log n/n)$ and $q \geq \log n/n^2$. Then $G(2n, p, q)$ is f-good w.h.p.*

Note that the proof of 1 is not difficult since $p = \omega(\log n/n)$ and $q \geq \log n/n^2$ [21]. Proving 2 and 3, however, is more challenging: see the full version of this paper [36].

From Theorem 9, $G(2n, p, q)$ is f^{Bo3}-good w.h.p. Hence, we consider the Best-of-three on an f^{Bo3}*-good* $G(2n, p, q)$. From 2 and 3, we have

$$\mathbf{E}[|A'_i|] = \sum_{v\in V_i} f^{\mathrm{Bo3}}\left(\frac{\deg_A(v)}{\deg(v)}\right) = nf^{\mathrm{Bo3}}\left(\frac{|A_i|p + |A_{3-i}|q}{n(p+q)}\right) \begin{cases} \pm O\left(\sqrt{\frac{n}{p}}\right) \\ +O\left(|A|\sqrt{\frac{\log n}{np}}\right) \end{cases} \tag{2}$$

for all $A \subseteq V$ and $i = 1, 2$. Here, we remark that 3 is stronger than 2 if $|A|$ is sufficiently small. This property will play a key role in the proof of Proposition 14.

Idea of the proof of Theorem 9

We consider the property 2. Note that we may assume $f(x) = x^k$ for some constant k w.l.o.g. since it suffices to obtain the concentration result for each term of f. For simplicity, let us exemplify our idea on the special case of $k = 3$. It is known that $\deg(v) = n(p+q) \pm O(\sqrt{np \log n})$ holds for all $v \in V$ w.h.p. (see, e.g. [21]). This implies that $\sum_{v \in S} \left(\frac{\deg_A(v)}{\deg(v)}\right)^3 = \frac{1 \pm O(\sqrt{\log n/np})}{(n(p+q))^3} \cdot \sum_{v \in S} \deg_A(v)^3$ holds for all $S, A \subseteq V$. Indeed, it is not difficult to see that the term $O(\sqrt{\log n/np})$ can be improved to $O(\sqrt{1/np})$.

The core of the proof is the concentration of $\sum_{v \in S} \deg_A(v)^3$. Note that $\sum_{v \in S} \deg_A(v) = \sum_{v \in S} \sum_{a \in A} \mathbb{1}_{\{s,a\} \in E}$ counts the number of cut edges between S and A. For fixed S and A, the Chernoff bound yields the concentration of it since each edge appears independently. Similarly, the summation $\sum_{v \in S} \deg_A(v)^3 = \sum_{v \in S} \sum_{a,b,c \in A} \mathbb{1}_{\{v,a\},\{v,b\},\{v,c\} \in E}$ counts the number of "crossing stars" between S and A. However, the Chernoff bound does not work here due to the dependency of the appearance of crossing stars. Fortunately, we can obtain a strong lower bound using the Janson inequality as follows: For $S, A, B, C \subseteq V$, let $W(S; A, B, C) := \sum_{v \in S} \deg_A(v) \deg_B(v) \deg_C(v)$. From the Janson inequality and the union bound on $S, A, B, C \subseteq V$, we can show that $W(S; A, B, C) \geq \mathbf{E}[W(S; A, B, C)] - O(n^{3.5} p^{2.5})$ holds for all $S, A, B, C \subseteq V$ w.h.p. On the other hand, it is easy to check that

$$\begin{aligned} W(S; A, B, C) = W(V; V, V, V) - W(V; V, V, V \setminus C) - W(V; V, V \setminus B, C) \\ - W(V; V \setminus A, B, C) - W(V \setminus S; A, B, C). \end{aligned}$$

The Kim-Vu concentration yields $W(V; V, V, V) \leq \mathbf{E}[W(V; V, V, V)] + O(n^{3.5} p^{2.5})$ since we do not consider the union bound here. For the other terms, we apply the lower bound by the Janson inequality. Then, we have a strong concentration result that $\sum_{v \in S} \deg_A(v)^3 = W(S; A, A, A) = \mathbf{E}[W(S; A, A, A)] \pm O(n^{3.5} p^{2.5})$ holds for all $S, A \subseteq V$ w.h.p. Finally, we estimate the gap between $\mathbf{E}[W(S \cap V_i, A, A, A)]$ and $|S \cap V_i|(|A_i|p + |A_{3-i}|q)^3$. See the full version [36] for details.

4 Induced dynamical system

Let $\alpha_i := \frac{|A_i|}{n}$, $\alpha_i' := \frac{|A_i'|}{n}$ and $r := \frac{q}{p}$. Suppose that r is a constant. Then, for an f^{Bo3}-good $G(2n, p, q)$, it holds w.h.p. that

$$\left| \alpha_i' - f^{\mathrm{Bo3}}\left(\frac{\alpha_i + r\alpha_{3-i}}{1+r}\right) \right| = O\left(\sqrt{\frac{1}{np}} + \sqrt{\frac{\log n}{n}}\right) \tag{3}$$

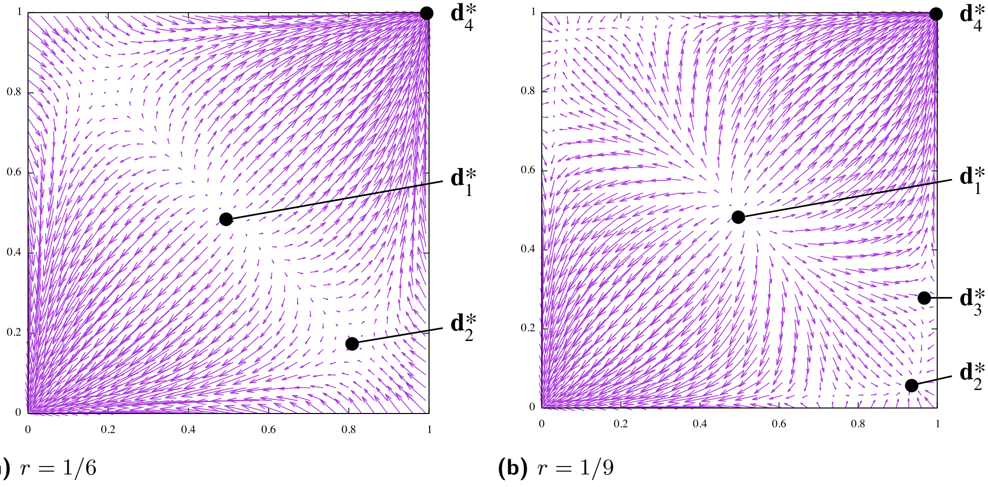
for all $A \subseteq V$ and $i = 1, 2$ since (1) and (2) hold.

Throughout this paper, we use $\boldsymbol{\alpha} = (\alpha_1, \alpha_2)$ and $\boldsymbol{\alpha}' = (\alpha_1', \alpha_2')$ as vector-valued random variables. Equation (3) leads us to the dynamical system H, where we define $H : \mathbb{R}^2 \to \mathbb{R}^2$ as

$$H : \mathbf{a} \mapsto (H_1(\mathbf{a}), H_2(\mathbf{a})), \tag{4}$$

and $H_i(a_1, a_2) := f^{\mathrm{Bo3}}\left(\frac{a_i + r a_{3-i}}{1+r}\right)$.

By combining (3) with the Lipschitz condition, it is not difficult to show the following result; see Section 6 for the proof.



(a) $r = 1/6$ **(b)** $r = 1/9$

Figure 2 The induced dynamical system H of (4). The points $\mathbf{d}_i^*$ are the fixed points given in (7). Here, the horizontal and vertical axes correspond to α_1 and α_2, respectively. We can observe two sink points in (b), but none in (a).

▶ **Theorem 10.** *Consider the Best-of-three on an f^{Bo3}-good $G(2n,p,q)$, starting with the vertex set $A^{(0)} \subseteq V$ holding opinion 1. Let $(\boldsymbol{\alpha}^{(t)})_{t=0}^{\infty}$ be a stochastic process given by $\boldsymbol{\alpha}^{(t)} = (\alpha_1^{(t)}, \alpha_2^{(t)})$ and $\alpha_i^{(t)} = |A^{(t)} \cap V_i|/n$. Let H be the mapping (4) and define $(\mathbf{a}^{(t)})_{t=0}^{\infty}$ as*

$$\begin{cases} \mathbf{a}^{(0)} = \boldsymbol{\alpha}^{(0)}, \\ \mathbf{a}^{(t+1)} = H(\mathbf{a}^{(t)}). \end{cases} \tag{5}$$

Then there exists a positive constant $C > 0$ such that

$$\forall 0 \leq t \leq n^{o(1)}, \forall A^{(0)} \subseteq V :$$
$$\mathbf{Pr}\left[\|\boldsymbol{\alpha}^{(t)} - \mathbf{a}^{(t)}\|_\infty \leq C^t \left(\frac{1}{\sqrt{np}} + \sqrt{\frac{\log n}{n}} \right) \right] \geq 1 - n^{-\Omega(1)}.$$

Broadly speaking, Theorem 10 approximates the behavior of $\boldsymbol{\alpha}^{(t)}$ by the orbit $\mathbf{a}^{(t)}$ of the corresponding dynamical system H. We call the mapping H the *induced dynamical system*. Indeed, the same results as (2) hold for the Best-of-two voting. Therefore, analogous results of Theorem 10 hold, which enable us to analyze the Best-of-two on $G(2n,p,q)$ via its induced dynamical system. The dynamical system H of (4) is illustrated in Figure 2.

To make the calculations more convenient, we change the coordinate of H by

$$\boldsymbol{\delta} = (\delta_1, \delta_2) := (\alpha_1 - \alpha_2, \alpha_1 + \alpha_2 - 1).$$

Note that δ_1 and δ_2 axes are corresponding to the dotted lines of Figure 1. Let $u := \frac{1-r}{1+r}$. Then we have

$$\mathbf{E}[\delta_i' \mid A] = T_i(\delta_1, \delta_2) + O\left(\frac{1}{\sqrt{np}}\right),$$

where

$$T_1(d_1, d_2) := \frac{ud_1}{2}\left(3 - (ud_1)^2 - 3d_2^2\right), \, T_2(d_1, d_2) := \frac{d_2}{2}\left(3 - 3(ud_1)^2 - d_2^2\right).$$

This suggests another dynamical system $T(\mathbf{d}) = (T_1(\mathbf{d}), T_2(\mathbf{d}))$. Here, we use $\mathbf{d} = (d_1, d_2)$ as a specific point and $\boldsymbol{\delta} = (\delta_1, \delta_2)$ as a vector-valued random variable. Consider $\boldsymbol{\delta}^{(t)} = (\delta_1^{(t)}, \delta_2^{(t)})$ and $(\mathbf{d}^{(t)})_{t=0}^{\infty}$, where $\mathbf{d}^{(0)} = \boldsymbol{\delta}^{(0)}$ and $\mathbf{d}^{(t+1)} = T(\mathbf{d}^{(t)})$ for each $t \geq 0$. From Theorem 10, it holds w.h.p. that

$$\|\boldsymbol{\delta}^{(t)} - \mathbf{d}^{(t)}\|_\infty \leq C^t \left(\frac{1}{\sqrt{np}} + \sqrt{\frac{\log n}{n}} \right) \tag{6}$$

for sufficiently large constant $C > 0$, any $0 \leq t \leq n^{o(1)}$ and any initial configuration $A^{(0)} \subseteq V$. For notational convenience, we use $\boldsymbol{\delta}' := \boldsymbol{\delta}^{(t+1)}$ for $\boldsymbol{\delta} = \boldsymbol{\delta}^{(t)}$. Similarly, we refer $\mathbf{d}'$ to $T(\mathbf{d})$.

Note that $\boldsymbol{\delta}$ satisfies $|\delta_1| + |\delta_2| \leq 1$. In addition, the dynamical system T is symmetric: Precisely, $T_1(\pm d_1, \mp d_2) = \pm T_1(d_1, d_2)$ and $T_2(\pm d_1, \mp d_2) = \mp T_2(d_1, d_2)$ hold. In Lemma 11, we assert that the sequence $(\mathbf{d}^{(t)})_{t=0}^{\infty}$ is closed in

$$S := \{(d_1, d_2) \in [0,1]^2 : d_1 + d_2 \leq 1\}.$$

From now on, we focus on S and consider the behavior of $\boldsymbol{\delta}$ around fixed points. A straightforward calculation shows that $\mathbf{d}' = \mathbf{d} \in S$ if and only if $\mathbf{d} \in \{\mathbf{d}_1^*, \mathbf{d}_2^*, \mathbf{d}_3^*, \mathbf{d}_4^*\}$, where

$$\mathbf{d}_i^* := \begin{cases} (0,0) & \text{if } i = 1, \\ \left(\sqrt{\frac{3u-2}{u^3}}, 0\right) & \text{if } i = 2 \text{ and } u \geq \frac{2}{3}, \\ \left(\sqrt{\frac{1}{4u^3}}, \sqrt{\frac{4u-3}{4u}}\right) & \text{if } i = 3 \text{ and } u \geq \frac{3}{4}, \\ (0,1) & \text{if } i = 4. \end{cases} \tag{7}$$

Here, we provide auxiliary results needed for the proofs of Theorems 2 and 4. Section 7 contains generalized form of these results.

For $\mathbf{x} \in \mathbb{R}^2$ and $\epsilon > 0$, let $B(\mathbf{x}, \epsilon) = \{\mathbf{y} \in \mathbb{R}^2 : \|\mathbf{x} - \mathbf{y}\|_\infty < \epsilon\}$ be the open ball. For $\mathbf{d} = (d_1, d_2) \in \mathbb{R}^2$, let $\langle \mathbf{d} \rangle_+ := (|d_1|, |d_2|) \in \mathbb{R}^2$.

▶ **Lemma 11** (S is closed)**.** *For any $\mathbf{d} \in S$, it holds that $\mathbf{d}' \in S$.*

▶ **Proposition 12** (Orbit convergence)**.** *For any sequence $(\mathbf{d}^{(t)})_{t=0}^{\infty}$, $\lim_{t\to\infty} \langle \mathbf{d}^{(t)} \rangle_+ = \mathbf{d}_i^*$ for some $i \in \{1,2,3,4\}$. In addition, if $u < \frac{3}{4}$ and a positive constant $\kappa > 0$ exists such that the initial point $\mathbf{d}^{(0)} = (d_1^{(0)}, d_2^{(0)}) \in S$ satisfies $|d_2^{(0)}| > \kappa$, then $\lim_{t\to\infty} \langle \mathbf{d}^{(t)} \rangle_+ = \mathbf{d}_4^*$.*

▶ **Proposition 13** (Dynamics around $\mathbf{d}_2^*$)**.** *Consider the Best-of-three on an f^{Bo3}-good $G(2n,p,q)$ such that $r = q/p < 1/7$ is a constant. Then there exists a positive constant $\epsilon = \epsilon(r)$ satisfying*

$$\mathbf{Pr}\left[\langle \boldsymbol{\delta}' \rangle_+ \notin B(\mathbf{d}_2^*, \epsilon) \,\middle|\, \langle \boldsymbol{\delta} \rangle_+ \in B(\mathbf{d}_2^*, \epsilon)\right] \leq \exp(-\Omega(n)).$$

In particular, $T_{\mathrm{cons}}(A) = \exp(\Omega(n))$ w.h.p. for any A satisfying $\langle \boldsymbol{\delta} \rangle_+ \in B(\mathbf{d}_2^*, \epsilon)$.

▶ **Proposition 14** (Towards consensus)**.** *Consider the Best-of-three on an f^{Bo3}-good $G(2n,p,q)$ such that $r = q/p$ is a constant. Then, there exists a universal constant $\epsilon = \epsilon(r) > 0$ satisfying the following: $T_{\mathrm{cons}}(A) \leq O(\log\log n + \log n/\log(np))$ holds w.h.p. for all $A \subseteq V$ with $\min\{|A|, 2n - |A|\} \leq \epsilon n$.*

▶ **Proposition 15** (Escape from fixed points)**.** *Consider the Best-of-three on an f^{Bo3}-good $G(2n,p,q)$ such that p and q are constants. If $q/p > 1/7$ and $|\delta_2^{(0)}| = o(1)$, then it holds w.h.p. that $|\delta_2^{(\tau)}| > \kappa$ for some $\tau = O(\log n)$ and some constant $\kappa > 0$.*

Intuitive explanations for Propositions 13 to 15

In Propositions 13 to 15, we consider the behavior of $\boldsymbol{\alpha}^{(t)}$ around the fixed points (7). Let H be the induced dynamical system and let J be the Jacobian matrix of H at a fixed point $\mathbf{a}^*$ with two eigenvalues λ_1, λ_2. If the eigenvectors are linearly independent, we can rewrite J as $J = U^{-1}\Lambda U$, where $\Lambda := \mathrm{diag}(\lambda_1, \lambda_2)$ and U is some nonsingular matrix. Let $\boldsymbol{\beta} := U(\boldsymbol{\alpha} - \mathbf{a}^*)$. Roughly speaking, if $\boldsymbol{\alpha}$ is closed to $\mathbf{a}^*$, the Taylor expansion at $\mathbf{a}^*$ (i.e. $H(\boldsymbol{\alpha}) \approx \mathbf{a}^* + J(\boldsymbol{\alpha} - \mathbf{a}^*)$) yields

$$\mathbf{E}[\boldsymbol{\beta}' \mid A] = U(\mathbf{E}[\boldsymbol{\alpha}' \mid A] - \mathbf{a}^*) \approx U(H(\boldsymbol{\alpha}) - \mathbf{a}^*) \approx \Lambda \boldsymbol{\beta}.$$

In other words, $\beta_i' \approx \lambda_i \beta_i$. If $\max\{|\lambda_1|, |\lambda_2|\} < 1 - c$ for some constant $c > 0$, we might expect that $\|\boldsymbol{\beta}\| = \Theta(\|\boldsymbol{\alpha} - \mathbf{a}^*\|)$ is likely to keep being small. Here, we do not restrict this argument on the Best-of-three. We will prove Proposition 19, which is a generalized version of Proposition 13. If $\max\{|\lambda_1|, |\lambda_2|\} > 1 + c$ for some constant $c > 0$, the norm $\|\boldsymbol{\beta}\|$ seems to become large in a small number of steps. We will exploit this insight and prove Proposition 25, which immediately implies Proposition 15. Indeed, for consensus areas (i.e. $\mathbf{a}^* \in \{(0,0), (1,1)\}$), the induced dynamical systems of the Best-of-three and the Best-of-two satisfy $\lambda_1 = \lambda_2 = 0$. Then, the Taylor expansion yields $\|\boldsymbol{\alpha}' - \mathbf{a}^*\| \approx O(\|\boldsymbol{\alpha} - \mathbf{a}^*\|^2)$. This observation and the property 3 lead to the proof of Proposition 20 as well as Proposition 14.

5 Derive Theorems 2 and 4

Here, we prove Theorems 2 and 4 using Propositions 12 to 15.

Proof of Theorem 2. If $r > \frac{1}{7}$ and $A^{(0)} \subseteq V$ satisfies $\big||A^{(0)}| - n\big| = \Omega(n)$, then we have $|d_2^{(0)}| = |\delta_2^{(0)}| > \kappa$ for some constant $\kappa > 0$. Next, for any constant $\epsilon > 0$, Proposition 12 implies $\langle \mathbf{d}^{(l)} \rangle_+ \in B(\mathbf{d}_4^*, \epsilon)$ for some constant $l = l(\epsilon)$. From (6), we have $\langle \boldsymbol{\delta}^{(l)} \rangle_+ \in B(\mathbf{d}_4^*, \epsilon)$ for sufficiently large n. Set ϵ be the constant mentioned in Proposition 14. Then, from Proposition 14, it holds w.h.p. that $T_{\mathrm{cons}}(A^{(0)}) \le l + T_{\mathrm{cons}}(A^{(l)}) \le O(\log\log n + \log n / \log(np))$.

If $r < \frac{1}{7}$, Proposition 13 yields $T_{\mathrm{cons}}(A^{(0)}) \ge \exp(\Omega(n))$ w.h.p. for any $A^{(0)} \subseteq V$ with $\boldsymbol{\delta}^{(0)} \in B(\mathbf{d}_2^*, \epsilon)$, where $\epsilon > 0$ is the constant from Proposition 13, which completes the proof of ii. ◀

Proof of Theorem 4. If $|\delta^{(0)}| = o(1)$, then Proposition 15 yields that $|\delta^{(\tau)}| > \kappa$ for some constant $\kappa > 0$ and some $\tau = O(\log n)$. Then, from Theorem 2, we have $T_{\mathrm{cons}}(A^{(\tau)}) \le O(\log\log n + \log n / \log(np))$. Thus, $T_{\mathrm{cons}}(A^{(0)}) \le \tau + T_{\mathrm{cons}}(A^{(\tau)}) \le O(\log n)$. ◀

6 Polynomial voting processes

Using Theorem 9, we can prove the same results as Theorem 10 for various models including the Best-of-two. Hence, in this paper, we do not restrict our interest to the Best-of-three: Instead, we prove general results that hold for *polynomial voting process* on $G(2n, p, q)$.

▶ **Definition 16** ((f_1, f_2)-polynomial voting process). *Let $G = (V, E)$ be a graph where each vertex holds an opinion from $\{1, 2\}$. Let $f_1, f_2 : [0,1] \to [0,1]$ be polynomials. For the set A of vertices with opinion 1, let A' denote the set of vertices with opinion 1 after an update. In the (f_1, f_2)-polynomial voting process, $A' = \{v \in V : X_v = 1\}$ where $(X_v)_{v \in V}$ are independent binary random variables satisfying*

$$\mathbf{Pr}[X_v = 1] = \begin{cases} f_1\left(\frac{\deg_A(v)}{\deg(v)}\right) & (v \in A, \text{ i.e. } v \text{ has opinion } 1) \\ f_2\left(\frac{\deg_A(v)}{\deg(v)}\right) & (v \in V \setminus A, \text{ i.e. } v \text{ has opinion } 2). \end{cases}$$

In other words, for $i = 1, 2$,

$$\mathbf{Pr}[v \in A' \mid A, v \text{ has opinion } i] = f_i\left(\frac{\deg_A(v)}{\deg(v)}\right).$$

Polynomial voting process includes several known voting models including the Best-of-two, the Best-of-three, and so on. For example, $f_1(x) = f_2(x) = f^{\mathrm{Bo3}}(x) = 3x^2 - 2x^3$ for the Best-of-three. For the Best-of-two, $f_1(x) = 2x(1-x)$ and $f_2(x) = x^2$. We can define induced dynamical system for any polynomial voting process on $G(2n, p, q)$ via the following result:

▶ **Theorem 17** (Theorem 10 for polynomial voting processes). *Let f_1 and f_2 be polynomials with constant degree. Consider an (f_1, f_2)-polynomial voting process, on an f_1-good and f_2-good $G(2n, p, q)$ starting with vertex set $A^{(0)} \subseteq V$ of opinion 1. Let $(A^{(t)})_{t=0}^{\infty}$ be a sequence of random vertex subsets defined by $A^{(t+1)} := (A^{(t)})'$ for each $t \geq 0$. Let $(\boldsymbol{\alpha}^{(t)})_{t=0}^{\infty}$, where $\boldsymbol{\alpha}^{(t)} = (\alpha_1^{(t)}, \alpha_2^{(t)})$ and $\alpha_i^{(t)} = |A^{(t)} \cap V_i|/n$. Define a mapping $H = (H_1, H_2)$ as*

$$H_i(a_1, a_2) = a_i f_1\left(\frac{a_i + r a_{3-i}}{1+r}\right) + (1 - a_i) f_2\left(\frac{a_i + r a_{3-i}}{1+r}\right) \quad \text{for } i = 1, 2.$$

Define $(\mathbf{a}^{(t)})_{t=0}^{\infty}$ as $\mathbf{a}^{(0)} = \boldsymbol{\alpha}^{(0)}$ and $\mathbf{a}^{(t+1)} = H(\mathbf{a}^{(t)})$ for each $t \geq 0$. Then, there exists a constant $C > 0$ such that

$$\forall 0 \leq t \leq n^{o(1)}, \forall A^{(0)} \subseteq V :$$
$$\mathbf{Pr}\left[\|\boldsymbol{\alpha}^{(t)} - \mathbf{a}^{(t)}\|_\infty \leq C^t\left(\frac{1}{\sqrt{np}} + \sqrt{\frac{\log n}{n}}\right)\right] \geq 1 - n^{-\Omega(1)}.$$

Remark that the mapping H of Theorem 17 is the induced dynamical system.

Proof. For any polynomial voting process, the cardinality $|A_i'| = \sum_{v \in V_i} X_v$ is the sum of independent random variables. Thus, if we fix $A \subseteq V$, the Hoeffding bound implies that (1) holds w.h.p. Since

$$\mathbf{E}[|A_i'|] = \sum_{v \in V_i} \mathbf{E}[X_v] = \sum_{v \in A_i} f_1\left(\frac{\deg_A(v)}{\deg(v)}\right) + \sum_{v \in V \setminus A_i} f_2\left(\frac{\deg_A(v)}{\deg(v)}\right),$$

the property 2 and (1) lead to

$$\|\boldsymbol{\alpha}' - H(\boldsymbol{\alpha})\|_\infty \leq C_1\left(\frac{1}{\sqrt{np}} + \sqrt{\frac{\log n}{n}}\right)$$

for some constant $C_1 > 0$.

Note that the function H satisfies the Lipschitz condition. Hence, a positive constant C_2 exists such that

$$\|H(\mathbf{x}) - H(\mathbf{y})\|_\infty \leq C_2 \|\mathbf{x} - \mathbf{y}\|_\infty$$

holds for any $\mathbf{x}, \mathbf{y} \in [0, 1]^2$. Let $\boldsymbol{\alpha}^{(t)} = (\alpha_1^{(t)}, \alpha_2^{(t)})$ be the vector-valued stochastic process and $\mathbf{a}^{(t)} = (\mathbf{a}_1^{(t)}, \mathbf{a}_2^{(t)})$ be the vector sequence given in (5). Then, we have

$$\begin{aligned}
\|\boldsymbol{\alpha}^{(t)} - \mathbf{a}^{(t)}\|_\infty &= \|\boldsymbol{\alpha}^{(t)} - H(\boldsymbol{\alpha}^{(t-1)}) + H(\boldsymbol{\alpha}^{(t-1)}) - H(\mathbf{a}^{(t-1)})\|_\infty \\
&\leq \|\boldsymbol{\alpha}^{(t)} - H(\boldsymbol{\alpha}^{(t-1)})\|_\infty + C_2 \|\boldsymbol{\alpha}^{(t-1)} - \mathbf{a}^{(t-1)}\|_\infty \\
&\leq C_2 \|\boldsymbol{\alpha}^{(t-1)} - \mathbf{a}^{(t-1)}\|_\infty + C_1\left(\frac{1}{\sqrt{np}} + \sqrt{\frac{\log n}{n}}\right) \\
&\leq C^t\left(\frac{1}{\sqrt{np}} + \sqrt{\frac{\log n}{n}}\right),
\end{aligned}$$

where C is sufficiently large constant. ◀

7 Results of general induced dynamical systems

Now let us focus on the orbit $(\boldsymbol{\alpha}^{(t)})_{t=1}^{\infty}$ such that $H(\boldsymbol{\alpha}^{(0)}) = \boldsymbol{\alpha}^{(0)}$ holds, where H is the induced dynamical system. In this case, Theorem 17 does not provide enough information about the dynamics. In dynamical system theory, a natural approach for the local behavior around fixed points is to consider the *Jacobian matrix*. Recall that, the Jacobian matrix J of a function $H : \mathbf{x} \mapsto (H_1(\mathbf{x}), H_2(\mathbf{x}))$ at $\mathbf{a} \in \mathbb{R}^2$ is a 2×2 matrix given by

$$J = \left(\frac{\partial H_i}{\partial x_j}(\mathbf{a}) \right)_{i,j \in [2]}.$$

In the following subsections, we will investigate the local dynamics from the viewpoint of the maximum singular value and eigenvalue of the Jacobian matrix.

In contrast to the local dynamics, it is quite difficult to predicate the orbit of general dynamical systems since some of them exhibits so-called chaos phenomenon. Therefore, the proof of the orbit convergence (e.g. Proposition 12) is not trivial. Fortunately, the induced dynamical system of the Best-of-three on $G(2n,p,q)$ is *competitive*, a well-known nice property for predicting the future orbit [26]. We can show Proposition 12 using known results of competitive dynamical systems. The same argument leads to the orbit convergence for the Best-of-two. Details are presented in the full version [36].

7.1 Sink point

We begin with defining the notion of sink points. Recall that the *singular value* of a matrix A is the positive square root of the eigenvalue of $A^\top A$.

▶ **Definition 18** (sink point). *For a dynamical system H, a fixed point $\mathbf{a}^* \in \mathbb{R}^2$ is* sink *if the Jacobian matrix J at $\mathbf{a}^*$ satisfies $\sigma_{\max} < 1$, where $\sigma_{\max}$ is the largest singular value of J.*

▶ **Proposition 19.** *Consider an (f_1, f_2)-polynomial voting process on an f_1-good and f_2-good $G(2n,p,q)$ such that $r = \frac{q}{p}$ is a constant. Let H be the induced dynamical system. Then, for any sink point $\mathbf{a}^*$ and any sufficiently small $\epsilon = \omega(\sqrt{1/np})$,*

$$\mathbf{Pr}\left[\boldsymbol{\alpha}' \notin B(\mathbf{a}^*, \epsilon) \,|\, \boldsymbol{\alpha} \in B(\mathbf{a}^*, \epsilon)\right] \le \exp(-\Omega(\epsilon^2 n))$$

holds. In particular, let

$$\tau := \inf\left\{ t \in \mathbb{N} : \boldsymbol{\alpha}^{(t)} \notin B(\mathbf{a}^*, \epsilon) \right\}$$

be a stopping time. Then, $\tau \ge \exp(\Omega(\epsilon^2 n))$ holds w.h.p. conditioned on $\boldsymbol{\alpha}^{(0)} \in B(\mathbf{a}^*, \epsilon)$ for any ϵ satisfying $\epsilon = \omega(\max\{1/\sqrt{np}, \sqrt{\log n/n}\})$.

7.2 Fast consensus

Suppose that the Jacobian matrix at the consensus point (i.e. $\alpha \in \{(0,0),(1,1)\}$ is the all-zero matrix. Then, we claim that the polynomial voting process reaches consensus within a small number of iterations if the initial set $A^{(0)}$ has small size.

▶ **Proposition 20.** *Consider an (f_1, f_2)-polynomial voting process on an f_1-good and f_2-good $G(2n,p,q)$ such that $\frac{p}{q}$ is a constant. Suppose that the Jacobian matrix at the point $\boldsymbol{\alpha} = (0,0)$ is the all-zero matrix. Then, there exists a constant $C_1, C_2, \delta > 0$ such that*

$$\mathbf{Pr}\left[T_{\mathrm{cons}}(A) \le C_1 \left(\log\log n + \frac{\log n}{\log np} \right) \right] \ge 1 - n^{-C_2}$$

for all $A \subseteq V$ satisfying $|A| \le \delta n$.

To show Proposition 20, we prove the following result which might be an independent interest:

▶ **Proposition 21.** *Consider a polynomial voting process on a graph G of n vertices. Suppose that there exist absolute constants $C, \delta > 0$ and a function $\epsilon = \epsilon(n) = o(1)$ such that*

$$\mathbf{E}[|A'|] \leq \frac{C|A|^2}{n} + \epsilon|A|$$

holds for all $A \subseteq V$ satisfying $|A| \leq \delta n$. Then, positive constants δ', C', C'' exist such that

$$\mathbf{Pr}\left[T_{\mathrm{cons}}(A) \leq C'\left(\log\log n + \frac{\log n}{\log \epsilon^{-1}}\right)\right] \geq 1 - n^{-C''}$$

holds for all $A \subseteq V$ satisfying $|A| \leq \delta' n$.

It should be noted that in Proposition 21, we do not restrict the underlying graph G to be random graphs.

7.3 Escape from a fixed point

Consider an (f_1, f_2)-polynomial voting process on an f_1-good and f_2-good $G(2n, p, q)$ such that p and q are constants. Let $\mathbf{a}^* \in \mathbb{R}^2$ be a fixed point of the induced dynamical system H. Let J be the Jacobian matrix of H at $\mathbf{a}^*$ and λ_1, λ_2 be its eigenvalues. Let $\mathbf{u}_i$ be the eigenvector of J corresponding to λ_i. Suppose that $\mathbf{u}_1, \mathbf{u}_2$ are linearly independent. Then, we can rewrite J as $J = U^{-1}\Lambda U$, where $\Lambda = \mathrm{diag}(\lambda_1, \lambda_2)$ and $U = (\mathbf{u}_1\, \mathbf{u}_2)^{-1}$. For a fixed point $\mathbf{a}^* \in \mathbb{R}^2$, let $\boldsymbol{\beta} = (\beta_1, \beta_2)$ be a vector-valued random variable defined as

$$\boldsymbol{\beta} = U(\boldsymbol{\alpha} - \mathbf{a}^*). \tag{8}$$

Roughly speaking, from the Taylor expansion of H at $\mathbf{a}^*$, we have

$$\mathbf{E}[\boldsymbol{\beta}'] \approx \Lambda\boldsymbol{\beta}$$

if $\|\boldsymbol{\beta}\|_\infty$ is sufficiently small. Thus, $|\beta'_i| \approx |\lambda_i||\beta_i|$.

Recall that $B(\mathbf{x}, R)$ is the open ball of radius R centered at $\mathbf{x}$. If $|\lambda_i| > 1$ for some $i \in [2]$, one may expect that $\boldsymbol{\alpha}^{(\tau)} \notin B(\mathbf{a}^*, \epsilon_0)$ holds for any $A^{(0)} \subseteq V$ and for some constant $\epsilon_0 > 0$. We aim to prove this under some assumptions.

▶ **Asspmption 22** (Basic assumptions)**.** *We consider an (f_1, f_2)-polynomial voting process on an f_1-good and f_2-good $G(2n, p, q)$ for constants $p \geq q \geq 0$. Let $\mathbf{a}^*$ be a fixed point and J be the corresponding Jacobian matrix satisfying*

(A1) *The eigenvectors $\mathbf{u}_1$ and $\mathbf{u}_2$ are linearly independent.*

(A2) *A positive constant ϵ_0 exists such that $\mathbf{Var}[\alpha'_i \mid A] \geq \Omega(n^{-1})$ for all $i \in \{1, 2\}$ and all $A \subseteq V$ of $\boldsymbol{\alpha} \in B(\mathbf{a}^*, \epsilon_0)$.*

(A3) *The matrix J contains an eigenvalue λ satisfying $|\lambda| > 1$.*

Under Assumption 22, we can define the random variable $\boldsymbol{\beta}$ of (8). Further, we put the following.

▶ **Asspmption 23.** *In addition to Assumption 22, we assume that there exists a positive constant ϵ^* satisfying the followings:*

(A4) *There exist two positive constants ϵ_1, C such that*

$$|\mathbf{E}[\beta_i' \mid A]| \geq (1+\epsilon_1)|\beta_i| - \frac{C}{\sqrt{n}}$$

holds for any $A \subseteq V$ of $\|\boldsymbol{\beta}\| \leq \epsilon^*$ and any $i \in [2]$ of $|\lambda_i| > 1$.

(A5) *For any $i \in [2]$ of $|\lambda_i| \leq 1$,*

$$\mathbf{Pr}[|\beta_i'| \leq \epsilon^* \mid |\beta_i| \leq \epsilon^*] \geq 1 - n^{-\Omega(1)}.$$

Sometimes, it might be not easy to check the conditions of Assumption 23. In this paper, we provide the following alternative condition which is easy to check:

▶ **Asspmption 24.** *In addition to Assumption 22, we assume the following:*

(A6) *The eigenvalues λ_1, λ_2 of J satisfies $|\lambda_i| \neq 1$ for all $i \in [2]$.*

Based on the assumptions, we prove the following result:

▶ **Proposition 25** (Escape from source and sink areas)**.** *Let $\mathbf{a}^*$ be a fixed point satisfying either Assumption 23 or 24. Then, there exist $\tau = O(\log n)$ and a constant $\epsilon' > 0$ such that the followings hold w.h.p.:*

(i) $\|\beta^{(\tau)}\|_\infty > \epsilon'$, *and*

(ii) $|\beta_j^{(\tau)}| \leq \epsilon'$ *for any $j \in [2]$ of $|\lambda_j| \leq 1$.*

References

1. E. Abbe. Community detection and stochastic block models: recent developments. *Journal of Machine Learning Research*, 18(177):1–86, 2018.
2. E. Abbe and C. Sandon. Recovering communities in the general stochastic block model without knowing the parameters. *In Proceedings of the 28th International Conference on Neural Information Processing Systems (NIPS)*, 1:676–684, 2015.
3. M. A. Abdullah and M. Draief. Global majority consensus by local majority polling on graphs of a given degree sequence. *Discrete Applied Mathematics*, 1(10):1–10, 2015.
4. Y. Afek, N. Alon, O. Barad, E. Hornstein, N. Barkai, and Z. Bar-Joseph. A biological solution to a fundamental distributed computing problem. *Science*, 331(6014):183–185, 2011.
5. P. Barbillon, S. Donnet, E. Lazega, and A. Bar-Hen. Stochastic block models for multiplex networks: an application to a multilevel network of researchers. *Journal of the Royal Statistical Society Series A*, 180(1):295–314, 2017.
6. L. Becchetti, A. Clementi, P. Manurangsi, E. Natale, F. Pasquale, P. Raghavendra, and L. Trevisan. Average whenever you meet: Opportunistic protocols for community detection. *In Proceedings of the 26th Annual European Symposium on Algorithms (ESA)*, 7:1–13, 2018.
7. L. Becchetti, A. Clementi, E. Natale, F. Pasquale, R. Silvestri, and L. Trevisan. Simple dynamics for plurality consensus. *Distributed Computing*, 30(4):293–306, 2017.
8. L. Becchetti, A. Clementi, E. Natale, F. Pasquale, and L. Trevisan. Stabilizing consensus with many opinions. *In Proceedings of the 27th annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 620–635, 2016.
9. L. Becchetti, A. Clementi, E. Natale, F. Pasquale, and L. Trevisan. Find your place: Simple distributed algorithms for community detection. *In Proceedings of the 28th annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 940–959, 2017.
10. P. Berenbrink, A. Clementi, R. Elsässer, P. Kling, F. Mallmann-Trenn, and E. Natale. Ignore or comply? On breaking symmetry in consensus. *In Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 335–344, 2017.

11 J. Chen and B. Yuan. Detecting functional modules in the yeast protein-protein interaction network. *Bioinformatics*, 22(18):2283–2290, 2006.

12 C. Cooper, R. Elsässer, H. Ono, and T. Radzik. Coalescing random walks and voting on connected graphs. *SIAM Journal on Discrete Mathematics*, 27(4):1748–1758, 2013.

13 C. Cooper, R. Elsässer, and T. Radzik. The power of two choices in distributed voting. *In Proceedings of the 41st International Colloquium on Automata, Languages, and Programming (ICALP)*, 2:435–446, 2014.

14 C. Cooper, R. Elsässer, T. Radzik, N. Rivera, and T. Shiraga. Fast consensus for voting on general expander graphs. *In Proceedings of the 29th International Symposium on Distributed Computing (DISC)*, pages 248–262, 2015.

15 C. Cooper, T. Radzik, N. Rivera, and T. Shiraga. Fast plurality consensus in regular expanders. *In Proceedings of the 31st International Symposium on Distributed Computing (DISC)*, 91(13):1–16, 2017.

16 C. Cooper and N. Rivera. The linear voting model. *In Proceedings of the 43rd International Colloquium on Automata, Languages, and Programming (ICALP)*, 55(144):1–12, 2016.

17 E. Cruciani, E. Natale, A. Nusser, and G. Scornavacca. Phase transition of the 2-choices dynamics on core-periphery networks. *In Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 777–785, 2018.

18 E. Cruciani, E. Natale, and G. Scornavacca. Distributed community detection via metastability of the 2-choices dynamics. *In Proceedings of the 33rd AAAI conference on artificial intelligence (AAAI)*, pages 6046–6053, 2019.

19 B. Doerr, L. A. Goldberg, L. Minder, T. Sauerwald, and C. Scheideler. Stabilizing consensus with the power of two choices. *In Proceedings of the 23rd annual ACM symposium on Parallelism in algorithms and architectures (SPAA)*, pages 149–158, 2011.

20 M. Fischer, N. Lynch, and M. Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1(1):26–39, 1986.

21 A. Frieze and M. Karoński. *Introduction to random graphs*. Campridge University Press, 2016.

22 M. Ghaffari and J. Lengler. Nearly-tight analysis for 2-choice and 3-majority consensus dynamics. *In Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 305–313, 2018.

23 S. Gilbert and D. Kowalski. Distributed agreement with optimal communication complexity. *In Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 965–977, 2010.

24 A. Goldenberg, A. X. Zheng, S. E. Fienberg, and E. M. Airoldi. A survey of statistical network models. *Foundations and Trends in Machine Learning*, 2(2):129–233, 2010.

25 Y. Hassin and D. Peleg. Distributed probabilistic polling and applications to proportionate agreement. *Information and Computation*, 171(2):248–268, 2001.

26 M. Hirsch and H. L. Smith. Monotone dynamical systems. *In Handbook of Differential Equations: Ordinary Differential Equations*, 2(4):239–357, 2005.

27 V. Kanade, F. Mallmann-Trenn, and T. Sauerwald. On coalescence time in graphs: When is coalescing as fast as meeting? *In Proceedings of the 30th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 956–965, 2019.

28 N. Kang and R. Rivera. Best-of-Three voting on dense graphs. *In Proceedings of the 31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 115–121, 2019.

29 J. H. Kim and V. H. Vu. Concentration of multivariate polynomials and its applications. *Combinatorica*, 20:417–434, 2000.

30 T. M. Liggett. *Interacting particle systems*. Springer-Verlag, 1985.

31 E. M. Marcotte, M. Pellegrini, H.-L. Ng, D. W. Rice, T. O. Yeates, and D. Eisenberg. Detecting protein function and protein-protein interactions from genome sequences. *Science*, 285(5428):751–753, 1999.

32 E. Mossel, J. Neeman, and O. Tamuz. Majority dynamics and aggregation of information in social networks. *Autonomous Agents and Multi-Agent Systems*, 28(3):408–429, 2014.

33 T. Nakata, H. Imahayashi, and M. Yamashita. Probabilistic local majority voting for the agreement problem on finite graph. *In Proceedings of the 5th Annual International Computing and Combinatorics Conference (COCOON)*, pages 330–338, 1999.

34 R. I. Oliveira and Y. Peres. Random walks on graphs: new bounds on hitting, meeting, coalescing and returning. *In Proceedings of the 16th Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*, pages 119–126, 2019.

35 G. Schoenebeck and F. Yu. Consensus of interacting particle systems on Erdős-Rényi graphs. *In Proceedings of the 29th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1945–1964, 2018.

36 N. Shimizu and T. Shiraga. Phase Transitions of Best-of-Two and Best-of-Three on Stochastic Block Models. *arXiv*, 2019. `arXiv:1907.12212`.

Distributed Data Summarization in Well-Connected Networks

Hsin-Hao Su
Boston College, MA, USA
suhx@bc.edu

Hoa T. Vu
Boston College, MA, USA
vuhd@bc.edu

Abstract

We study distributed algorithms for some fundamental problems in data summarization. Given a communication graph G of n nodes each of which may hold a value initially, we focus on computing $\sum_{i=1}^{N} g(f_i)$, where f_i is the number of occurrences of value i and g is some fixed function. This includes important statistics such as the number of distinct elements, frequency moments, and the empirical entropy of the data.

In the CONGEST model, a simple adaptation from streaming lower bounds shows that it requires $\tilde{\Omega}(D+n)$ rounds, where D is the diameter of the graph, to compute some of these statistics exactly. However, these lower bounds do not hold for graphs that are well-connected. We give an algorithm that computes $\sum_{i=1}^{N} g(f_i)$ exactly in $\tau_G \cdot 2^{O(\sqrt{\log n})}$ rounds where τ_G is the mixing time of G. This also has applications in computing the top k most frequent elements.

We demonstrate that there is a high similarity between the GOSSIP model and the CONGEST model in well-connected graphs. In particular, we show that each round of the GOSSIP model can be simulated almost perfectly in $\tilde{O}(\tau_G)$ rounds of the CONGEST model. To this end, we develop a new algorithm for the GOSSIP model that $1 \pm \epsilon$ approximates the p-th frequency moment $F_p = \sum_{i=1}^{N} f_i^p$ in $\tilde{O}(\epsilon^{-2} n^{1-k/p})$ rounds [1], for $p \geq 2$, when the number of distinct elements F_0 is at most $O\left(n^{1/(k-1)}\right)$. This result can be translated back to the CONGEST model with a factor $\tilde{O}(\tau_G)$ blow-up in the number of rounds.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Networks → Network algorithms; Mathematics of computing → Graph algorithms

Keywords and phrases Distributed Algorithms, Network Algorithms, Data Summarization

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.33

Related Version A full version of the paper is available at `https://arxiv.org/abs/1908.00236`.

1 Introduction

Motivation. Analyzing massive datasets has become an increasingly important and challenging problem. Collecting the entire data to one single machine is usually infeasible due to memory, I/O, or network bandwidth constraints. Furthermore, in many cases, data are distributed over the network and we hope to aggregate some of their properties efficiently. In this work, we consider several fundamental data summarization problems in distributed networks, specifically in the CONGEST and GOSSIP models.

In this problem, we have a graph $G = (V, E)$ of n nodes. Each node v in the graph may hold a value $\mathrm{val}(v)$ in the range $\{1, \dots, N\} \cup \{\text{NULL}\}$ where NULL simply means that the node does not hold a value. If $\mathrm{val}(v) = \text{NULL}$, we call v an *empty node*.

[1] $\tilde{O}$ omits polylog(n) factors.

© Hsin-Hao Su and Hoa T. Vu;
licensed under Creative Commons License CC-BY

We often use the notation $[N] := \{1, \ldots, N\}$. Let f_i be the number of nodes that hold value i, i.e., $f_i = |\{v \in V : \mathrm{val}(v) = i\}|$. We want to compute $\sum_{i=1}^{N} g(f_i)$ for some fixed function g. To demonstrate some important cases, consider the following examples.

Consider $g(f_i) = 1$ if $f_i > 0$ and 0 otherwise. This corresponds to the problem of counting the number of distinct elements (or computing the 0-th frequency moment F_0). The problem may arise in the following situation: Each node stores a version of a file (e.g. the hash of a blockchain), and we want to know how many different versions there are in the network.

If $g(f_i) = f_i^p$ for some fixed $p = 2, 3, \ldots$, then this corresponds to the problem of computing the p-th frequency moment F_p. We note that F_p is a basic, yet very important statistic of a dataset. F_2 measures the variance and could be used to estimate the size of a self-join in database applications. For higher p, F_p measures the skewness of the dataset (see [2]). Note that F_1 can be computed in $O(D)$ rounds in the CONGEST model by aggregating along a breath-first-search (BFS) tree (in the GOSSIP model F_1 can be computed exactly in $O(\log n)$ rounds).

Another example is $g(f_i) = -(f_i/F_1) \cdot \log(f_i/F_1)$. In this case, the sum is the empirical entropy of the data. Computing the empirical entropy is motivated by network applications such as detecting anomalies [20, 40, 42].

Models. We now give a formal description of the CONGEST and GOSSIP models, where the running time of an algorithm is measured by the number of rounds.

▶ **Definition 1.** *In the* CONGEST *model, we are given a graph $G = (V, E)$ of n nodes, in each synchronous round, each node can talk (send and receive message) to each of its neighbors and then perform local computations. Each message is restricted to be at most $O(\log n)$ bits.*

▶ **Definition 2.** *In the* GOSSIP(λ) *model with n nodes, in each synchronous round, each node u samples a node $t(u)$ from a distribution that satisfies the following: For any node v and any subset of nodes Z where $u \notin Z$,*

$$\Pr\left(t(u) = v \,\middle|\, \bigwedge_{z \in Z} t(z)\right) \in \left[\frac{1-\lambda}{n}, \frac{1+\lambda}{n}\right].$$

In the above, "$\bigwedge_{z\in Z} t(z)$" means conditioning on any assignment of each $t(z)$ for $z \in Z$. Then, u can PUSH a message of size $O(\log n)$ to $t(u)$ or PULL a message of size $O(\log n)$ from $t(u)$. Then, after performing some local computations, it proceeds to the next round. We refer GOSSIP *model as the* GOSSIP(0) *model.*

1.1 Our results

We organize our main results into three categories: a) results in the CONGEST model, b) an emulation of the GOSSIP model in the CONGEST model, and c) results in the GOSSIP model.

Results in the CONGEST model. We briefly show how to adapt streaming algorithms to approximate F_p (for $p = 0, 2, 3, \ldots$) in the CONGEST model. We also demonstrate some lower bounds and conditional lower bounds that give evidence that such algorithms are optimal or near-optimal.

The lower bounds show that computing F_p exactly for $p = 0, 2, 3, \ldots$ requires $\tilde{\Omega}(D + n)$ rounds and approximating F_p within a constant factor requires *polynomial* rounds in n for $p \geq 3$. Roughly speaking, the hard instances in the CONGEST model are graphs with a small

balanced cut of $O(1)$ size that causes an information bottleneck. However, such bottleneck does not occur in graphs that are well-connected. Our first main result aims to answer the following question: *Could one design more efficient algorithms for well-connected graphs?* We give a positive answer to this question.

By using the permutation routing algorithms of Ghaffari et al. [15] (later improved by Ghaffari and Li [16]), we show that there exists an algorithm running in $\tau_G \cdot 2^{O(\sqrt{\log n})}$ rounds that computes $\sum_{i=1}^{N} g(f_i)$ for all fixed and computable functions g with high probability (w.h.p.) [2]. This includes all the aforementioned quantities such as the number of distinct elements, higher frequency moments, and the empirical entropy. Thus, if the graph has small mixing time such as expanders [19, 23], where $\tau_G = \text{polylog}(n)$, then we obtain a much more efficient sub-polynomial in n algorithm compared to the adaptation of the streaming counterpart.

▶ **Theorem 3** (Main result 1). *There exists an algorithm that computes $\sum_{i=1}^{N} g(f_i)$ exactly for all (fixed and computable) functions g in the* CONGEST *model in $\tau_G \cdot 2^{O(\sqrt{\log n})}$ rounds w.h.p.*

Our algorithm can also easily be extended to find the top k frequent elements in $O(k) + \tau_G \cdot 2^{O(\sqrt{\log n})}$ rounds.

From CONGEST to GOSSIP. The lower bounds do not apply directly to the GOSSIP model either. This is because for any balanced cut of the nodes, one expects $O(n)$ messages to be sent across in one round. Moreover, the expected communication degree per node in the GOSSIP model is $O(1)$. Intuitively, the graph formed by the communication pattern in the GOSSIP model is similar to an expander graph.

In fact, we show that well-connected graphs can emulate the GOSSIP model efficiently. In particular, one round of the GOSSIP $(1/\text{poly}(n))$ model can be emulated in $\tau_G \cdot \text{polylog}(n)$ rounds in the CONGEST model where the underlying graph is G. Therefore, any algorithm that works in the GOSSIP $(1/\text{poly}(n))$ model can be turned into an algorithm in the CONGEST model with an $\tilde{O}(\tau_G)$ factor blow-up.

Consider our results in the CONGEST model. The permutation routing algorithms of [15] and [16] introduce a super-logarithmic factor, $2^{O(\sqrt{\log n})}$, on top of the mixing time. It becomes the bottleneck in graphs with small mixing times (e.g., expanders). Improving the permutation routing algorithm directly yields improvements to our results in the CONGEST model (and many other problems). However, it is unclear if it can be improved. This emulation result serves as an alternative route to circumvent the $2^{O(\sqrt{\log n})}$ factor, if one develops efficient GOSSIP algorithms.

▶ **Theorem 4** (Main result 2). *For $\lambda = 1/\text{poly}(n)$, one round of the* GOSSIP(λ) *model can be emulated in $\tilde{O}(\tau_G)$ rounds in the* CONGEST *model where G is the connected graph denoting the underlying network.*

We believe that this emulation result may be of independent interest. Jelasity et al. [27] studied how to implement the gossip-based peer sampling service empirically. Our result is an additional way to implement the service with theoretical guarantees.

[2] We consider $1 - 1/\text{poly}(n)$ as high probability.

Table 1 Results summary for computing frequency moments F_p. (*) can also be used to compute $\sum_i g(f_i)$ for all fixed and computable functions g.

	Number of rounds	Assumption	Approximation
CONGEST	$\tau_G \cdot 2^{O(\sqrt{\log n})}$ (*)		Exact
	$O(\epsilon^{-2}\tau_G \cdot \text{polylog}\, n)$	$F_0 \leq O(n^{1/(p-1)})$	$1 \pm \epsilon$
GOSSIP	$O(\epsilon^{-2} \cdot n^{1-k/p} \cdot \text{polylog}\, n)$	$F_0 \leq O(n^{1/(k-1)})$	$1 \pm \epsilon$

Results in the GOSSIP model. Motivated by our emulation result, we develop algorithms for the GOSSIP model. In particular, we are interested in the following question: *Suppose the number of non-empty nodes are sublinear in n. Could we take advantage of the computational power of the empty nodes?*

Suppose that the number of non-empty nodes is at most $O(n^{1/(k-1)})$ (or more generally, $F_0 \leq O(n^{1/(k-1)})$). We show that for any $p \geq 2$, F_p can be approximated within a $1 \pm \epsilon$ factor in $O(\epsilon^{-2} n^{1-k/p} \log^2 n)$ rounds with high probability.

▶ **Theorem 5** (Main result 3). *If $F_0 = O(n^{1/(k-1)})$ for some integer $2 \leq k \leq p$, then there exists an algorithm that approximates F_p up to a $1 \pm \epsilon$ factor in $O(\epsilon^{-2} n^{1-k/p} \log^2 n)$ rounds in the* GOSSIP $(1/n^c)$ *model, for some sufficiently large constant c, w.h.p.*

The GOSSIP $(1/n^c)$ model will incur a $\pm 1/\text{poly}(n)$ additive error which we consider insignificant. Since $F_0 \leq n$, we have an algorithm that approximates F_2 in $\tilde{O}(\epsilon^{-2})$ rounds by setting $k = 2$. When $k > 2$, the empty nodes serve as the extra computation power to solve the problem. In such scenarios, we are able to obtain running time that is not known to be achievable by adapting the streaming counterpart. For example, when $k = 3$, $F_0 = O(n^{1/2})$, we may approximate F_3 within a constant factor in polylog(n) rounds. Direct adaption of known streaming algorithms [2, 3, 36] requires super-logarithmic rounds, even in the case where $F_0 = O(n^{1/2})$.

Combining Theorem 5 and Theorem 4 with $k = p$, we have the following corollary.

▶ **Corollary 6.** *If $F_0 = O(n^{1/(p-1)})$, then there exists an algorithm in the* CONGEST *model that approximates F_p up to a $1 \pm \epsilon$ factor in $\tilde{O}(\epsilon^{-2} \cdot \tau_G)$ rounds w.h.p.*

1.2 Related work and preliminaries

Related work. In the distributed setting, Kuhn et al. [33] studied the problem of finding the mode, i.e., the most frequent element, in the CONGEST model. Let D is the diameter of the graph, and f^* is the largest number of occurrences among the values. They gave an algorithm that uses $O(D + F_2/f^* \cdot \log F_0)$ rounds. They also briefly explained how to implement streaming algorithms for approximating F_0 and F_2 in the CONGEST models. Also related to data summarization, Kuhn et al. [34] designed selection algorithms in the CONGEST model.

In the data stream model, each stream token (i, x) corresponds to the update $f_i \leftarrow f_i + x$. The problem of approximating the number of distinct elements F_0 and frequency moments F_p have been extensively studied. An incomplete list includes [2–4, 14, 18, 24, 25, 29, 41]. Roughly speaking, the space complexity for approximating F_p in the data stream model is $\tilde{O}(\epsilon^{-2})$ for $0 \leq p \leq 2$ and $\tilde{O}(\epsilon^{-2} n^{1-2/p})$ for $p \geq 2$. Furthermore, it is known that approximating F_∞ (or identifying the mode) is not possible in sublinear space. In the data stream model, researchers have also studied the problem of approximating the entropy [9, 10, 22].

We will briefly discuss the similarities between the data stream model and the CONGEST model. Roughly speaking, since streaming algorithms use little memory, they can be adapted to the CONGEST model by passing the memory state of the corresponding algorithm along the breadth-first-search tree. Similarly, lower bounds from streaming algorithms literature can also be translated into lower bounds in the CONGEST model. Data aggregation problems have also been studied in directed networks [35].

There is also a rich literature in the GOSSIP model started by the work of [12]. Some examples include spreading message [13,30,38], computing the sum and average [11,31,32], renaming [17], and quantile computation [21].

Preliminaries. We introduce basic notations and algorithmic building blocks in the CONGEST model.

To ease our presentation, we assume $N = O(\text{poly}(n))$. In our algorithms, we often want to learn about the sum of all the values (or hash values, indicator variables) held by the nodes; this can be done in $O(D)$ rounds. Another algorithmic primitive, based on downcasts and upcasts, is to broadcast the k smallest values in $O(D+k)$ rounds.

We define the mixing time similarly to [15]. A lazy random walk is a random walk in which at each step, we stay at the same node with probability 0.5 and move to a random neighbor with probability 0.5. Lazy random walk ensures the existence of a unique stationary distribution (i.e., the walk is aperiodic). From now on, we simply refer to a lazy random walk as a random walk.

Let $P_u^t = (P_u^t(v_1), \ldots, P_u^t(v_n)) \in [0,1]^n$ denotes the probability distribution on the nodes after t steps of a lazy random walk that starts at u. A crucial property of a random walk is that it will converge to the stationary distribution $(\deg(v_1)/2m, \ldots, \deg(v_n)/2m)$. Define the mixing time τ_G to be the minimum t such that for any starting node u and any node v_i,

$$\left| P_u^t(v_i) - \frac{\deg(v_i)}{2m} \right| \leq \frac{\deg(v_i)}{2mn} .$$

Using an $O(D)$-round pre-proscessing, we can assume that each node has a unique ID in $[n]$. Suppose we want the nodes in a graph to have unique IDs in $[n]$. We can elect a leader and build a breadth-first-search (BFS) tree that is rooted at the leader in $O(D)$ rounds [37]. Each node u can learn about the number of nodes in T_v where v is a child of u and T_v is the subtree that is rooted at v. This is done by aggregating the size from the leaves upward. It is then straightforward to assign the IDs to the nodes based on the depth-first-search (DFS) ordering. Specifically, the root notifies each of its children v the range of the IDs in T_v, based on the DFS ordering, and then recurse on T_v. From now on, we can refer to the nodes by their IDs, i.e., $\text{ID}(v) = v$.

We will also make use of hash functions. An $O(1)$-wise independent hash function $h : [a] \to [b]$ where a and b are at most $\text{poly}(n)$ can be stored in $O(\log n)$ bits. Hence, if we need to use a hash function, a leader can broadcast such hash function (using a BFS tree) in $O(D + \log n)$ rounds in the CONGEST model and $O(\log n)$ rounds in the GOSSIP model.

2 Algorithms in the CONGEST Model

2.1 Approximation algorithms

Upper bounds. We show that we can adapt the streaming algorithms given by Bar-Yossef et al. [4] (for approximating F_0) and by Alon et al. [2] (for approximating F_p, where $p \geq 2$) to the CONGEST model (see the appendix in our full version [39]). This is not of particular

novelty though we need some careful pipelining arguments to optimize the number of rounds. Kuhn et al. [33] also briefly outlined similar results. However, the exact round-complexity for a good approximation w.h.p. is not very clear from their paper.

▶ **Theorem 7.** *There exists an $O(D + \epsilon^{-2} \log n)$-round algorithm in the* CONGEST *model that computes a $1 \pm \epsilon$ approximation of F_0 and F_2 w.h.p. Furthermore, for $p > 2$, there exists an $O(D + \epsilon^{-2} \min(n, N)^{1-1/p} \log n)$-round algorithm that computes a $1 \pm \epsilon$ approximation of F_p w.h.p.*

Lower bounds. We show that the dependence on ϵ is tight via a conditional lower bound. Moreover, computing F_p exactly requires $\tilde{\Omega}(n)$ rounds. The lower bounds are obtained by adapting the existing streaming lower bounds to the CONGEST model. Due to space constraint, we refer to the appendix in our full version for the discussion.

▶ **Theorem 8.** *We have the following lower bounds in the* CONGEST *model.*
- *If the conjecture in [8] holds, then approximating F_p (for fixed $p \neq 1$) up to a $1 \pm \epsilon$ factor requires $\Omega(D + \epsilon^{-2}/\log n)$ rounds.*
- *A (1 ± 0.1)-approximation of F_p, for $p > 2$, requires $\Omega\left(D + \left(N^{1-\frac{2}{p}} + n^{\frac{1-2/p}{1+1/p}}\right)/\log n\right)$ rounds.*
- *Computing F_p exactly requires $\Omega(D + n/\log n)$ rounds.*

Hence, we cannot expect a sublinear algorithms (in terms of N, n) when $\epsilon \ll 1/\sqrt{n}$ or when we want to obtain the exact answer. The lower bounds arise in graphs with a small balanced cut which causes an information bottleneck. This observation motivates us to design an exact algorithm when the graph is well-connected.

2.2 An exact algorithm in near mixing-time

In this subsection, we show that it is possible to beat the lower bounds and achieve an exact algorithm in sublinear time if the graph has fast mixing time. For example, expander graphs are sparse and have $O(\text{polylog}\, n)$ mixing time.

Suppose each node has a set of messages (of size $\text{polylog}(n)$) each of which has a destination that is another node. In parts of our algorithms, we want to route messages in a small number of rounds. We rely on the following routing algorithm in the CONGEST model that uses $\tau_G \cdot 2^{O(\sqrt{\log n})}$ rounds. We note that $2^{O(\sqrt{\log n})}$ is more than $\text{polylog}\, n$ but smaller than any n^ϵ for $\epsilon > 0$. Also note that $D = O(\tau_G)$. Let $\deg(v)$ be the degree of v in G.

▶ **Theorem 9** ([16], [15]). *If each node of G is the source and the destination of at most $\deg(v) \cdot 2^{O(\sqrt{\log n})}$ messages, then there is a randomized algorithm in the* CONGEST *model that delivers all the messages in $\tau_G \cdot 2^{O(\sqrt{\log n})}$ rounds w.h.p.*

We also rely on the idea of sorting networks. Recall that we refer to the nodes by their unique IDs in $[n]$. In a sorting network, in each step r, the sorting network will pick a set of disjoint pairs of nodes. We use $\text{val}(x, r)$ to denote the value that node x holds in the beginning of step r. For each pair x and y (where $x < y$) that is picked, x will keep the smaller value $\min(\text{val}(x, r), \text{val}(y, r))$ and y will keep the larger value $\max(\text{val}(x, r), \text{val}(y, r))$. We treat NULL as $-\infty$. The sorting network can be constructed, solely based on n, so that after $t = O(\log n)$ steps, the values are sorted [1]. That is if $x < y$, then $\text{val}(x, t) \leq \text{val}(y, t)$.

In the CONGEST model, each node can generate the sorting network (note that the construction of the sorting network is independent of the topology of G and the values held by the nodes). Furthermore, each step can be simulated by invoking Theorem 9. Thus, we have the following.

▶ **Lemma 10.** *In the* CONGEST *model, we can sort the nodes' values in* $\tau_G \cdot 2^{O(\sqrt{\log n})}$ *rounds w.h.p.*

We now complete the proof of our first main result.

Proof of Theorem 3. We now use $\mathrm{val}(v)$ to refer to the value that v holds after sorting. We say a node v is a *head* or a *tail* if $\mathrm{val}(v) \neq -\infty$ and its ID is the smallest or the largest respectively among the IDs of the nodes that hold the value $\mathrm{val}(v)$. A node v can tell that if it is a head or a tail by checking with the nodes $v+1$ and $v-1$ respectively using the routing algorithm in Theorem 9. We use $\mathrm{head}(i)$ and $\mathrm{tail}(i)$ to denote the IDs of the head and the tail of value i respectively.

Now, every node that is not a head or a tail marks its value as $-\infty$. Each remaining node forms a token consisting of its value, ID, and whether if it is a head or a tail (or both). We then use sorting networks again to sort the values in the graph. We will also swap the tokens if two nodes swap their values. Afterward, the head and the tail tokens of a value i will be at some two nodes v and $v+1$ (or just at a node v if $f_i = 1$). To this end, each node v that holds a head token (that is not also a tail token) with value i will check with nodes $v+1$ and $v-1$, using the routing algorithm, to collect $\mathrm{tail}(i)$ since either $v+1$ or $v-1$ must have the tail token of i. Now, v can compute $g(f_i) = g(\mathrm{tail}(i) - \mathrm{head}(i) + 1)$ and set this as its value. All the nodes that do not hold a head token set their values to 0. We then compute $\sum_{i=1}^{N} g(f_i)$ using the BFS tree in $O(D)$ rounds. ◀

The algorithm above is more robust compared to the AMS sketch since it can handle all fixed and computable functions g. The AMS sketch cannot guarantee sublinear space in the streaming model (or sublinear time in the CONGEST model) for many functions [5–7]. The above algorithm also immediately leads to an algorithm that finds the top k frequent elements.

Finding the top k frequent elements. At the end of the above algorithm, the occurrence of each value i is held by some node v. Recall we can find the top k elements in the graph using $O(D+k)$ rounds via upcasts. This immediately leads to the following result.

▶ **Theorem 11.** *There exists an algorithm that finds the top k elements (along with their occurrences) in the* CONGEST *model in* $O(k + \tau_G \cdot 2^{O(\sqrt{\log n})})$ *rounds w.h.p.*

3 Emulation of GOSSIP Model in the CONGEST Model

In Section 2, we have shown that the moments can be computed exactly in $\tau_G \cdot 2^{O(\sqrt{\log n})}$ rounds. If the permutation routing algorithm can be improved to $\mathrm{polylog}(n)$ rounds, then the running time of our algorithms would be improved to $\tilde{O}(\tau_G)$ rounds. Whether the $2^{O(\sqrt{\log n})}$ factor can be improved to $\mathrm{polylog}(n)$ is an intriguing open question.

Instead of tackling the complexity of permutation routing, in this section, we show that one round of the GOSSIP model can be emulated almost-perfectly in $\tilde{O}(\tau_G)$ rounds in the CONGEST model. Therefore, if there is a $\mathrm{polylog}(n)$-round algorithm in the GOSSIP model, it implies a $\tilde{O}(\tau_G)$ rounds algorithm in the CONGEST model. In Section 4, we present efficient algorithms in the GOSSIP model when F_0 is small (or when the number of empty nodes is large) which can be translated back to the CONGEST model using the emulation result in this section.

Recall that $P^t_u = (P^t_u(v_1), \dots, P^t_u(v_n)) \in [0,1]^n$ denotes the probability distribution on the nodes after t steps of a lazy random walk that starts at u (see Section 1.2). Given λ, we let $\tau_G(\lambda)$ be the smallest t such that for any starting node u and any node v_i,

$$\left| P^t_u(v_i) - \frac{\deg(v_i)}{2m} \right| \le \lambda.$$

Note that if $\lambda = 1/\operatorname{poly}(n)$ then $\tau_G(\lambda) = O(\tau_G)$ [15, Definition 2.1].

We will run several random walks in parallel. The following lemma from [15] shows that the parallel random walks can be performed efficiently in the CONGEST model.

▶ **Lemma 12** ([15], Lemma 2.5). *Let $G = (V, E)$ be an n-node graph and let $t \ge 1$ be a positive integer. Assume that we perform $T = O(\operatorname{poly}(n))$ steps of a collection of independent random walks in parallel. If each node $u \in V$ is the starting node of at most $t \cdot \deg(u)$ random walks, w.h.p., the T steps of all the random walks can be performed in $O((t + \log n) \cdot T)$ rounds in the* CONGEST *model.*

The main technical difficulty of the emulation lies in the fact that the stationary distribution is not necessarily uniform in general graphs. If G is regular, we could let each node u start a random walk that runs for $O(\tau_G)$ steps. The probability that u ends at each node is (nearly) uniform. If it ends at v then we set $t(u) = v$. Moreover, by Lemma 12, all the random walks can be performed simultaneously in $\tilde{O}(\tau_G)$ rounds.

In irregular graphs, such approach does not work because the stationary distribution is not uniform. One remedy is to regularize the random walk (i.e. adding self-loops to non-maximum degree nodes). However, this may significantly increase the mixing time of the graph (e.g., a star graph). In the following, we give an emulation algorithm whose running time is within a $\operatorname{polylog}(n)$ factor of the mixing time.

For each node u in G, we split it into $\deg(u)$ *compartments*. When a random walk enters a node, it is assigned randomly to one of its compartments. There are $2m$ compartments in G in total. We outline the emulation algorithm below.

1. Let $k = \lfloor 1.5m/n \rfloor$. Each node creates k *destination tokens* and distributes them over the compartments in G so that each compartment contains at most *one* destination token. Now $n \cdot k \approx 1.5m$ compartments are filled with tokens.
2. Each node sends out a *source token*. Each source token starts a random walk to distribute itself randomly over the compartments at the end. If the source token of node u ends in a compartment with the destination token of some node v, we set $t(u) = v$.
3. Route the message between u and $t(u)$ for each u simultaneously.

We explain how to implement each step in details.

Step 1. Each node u creates a destination token (u, k) initially. The first component of the token is its identity while the second component of the token is its multiplicity. The goal is to split the tokens and distribute them across the compartments so that all tokens have multiplicity of 1 and each compartment holds at most one token. We divide Step 1 into the **splitting phase** and the **distributing phase**.

The splitting phase is further divided into $\lceil \log k \rceil$ stages. At the beginning of each stage, if $W > 1$, each token (u, W) is split into two tokens $(u, \lceil W/2 \rceil)$ and $(u, \lfloor W/2 \rfloor)$. Then all tokens perform τ_G steps of random walks.

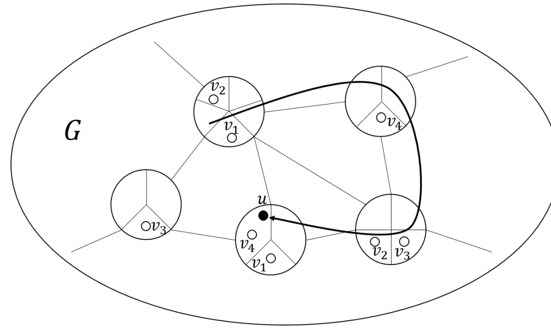


Figure 1 Illustration of Step 2. The random walk of u's source token ends in the compartment containing the destination token of v_4. Thus, $t(u) = v_4$.

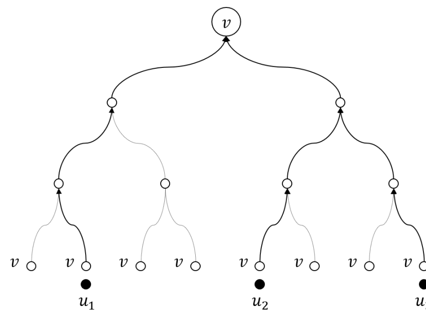


Figure 2 Illustration of Step 3. u_1, u_2, u_3 will follow the paths taken by the destination tokens of v back to v. The paths may overlap. W.h.p. every edge is contained in at most $O(\log n)$ paths.

We show that w.h.p., there are at most $O(\log n)$ tokens per compartment at the end of each stage. Given a stage, the probability that a token ends up in a given compartment in node v is at most

$$\left(\frac{\deg(v)}{2m} + \frac{1}{2mn}\right) \cdot \frac{1}{\deg(v)} \le \frac{1}{m}.$$

Since there are at most $k \cdot n \le 1.5m$ tokens, there are at most $O(1)$ tokens ending in a compartment in expectation. By standard Chernoff and union bound argument, w.h.p. there are at most $O(\log n)$ tokens in each compartment.

Moreover, since each node u holds at most $\deg(u) \cdot O(\log n)$ tokens at the beginning of each stage, the random walks can be performed in parallel in $O(\tau_G \cdot \log n)$ rounds by Lemma 12. Therefore, the splitting phase uses $O((\log k) \cdot (\tau_G \cdot (\log n))) = \tilde{O}(\tau_G)$ rounds. At the end of the splitting phase, the multiplicity of each token is one. Moreover, w.h.p. each compartment contains at most $O(\log n)$ tokens.

In the distributing phase, a compartment containing more than one token will start the random walks on *all except one* of its token for $\tau_G(0.1/2m)$ steps. Again, by Lemma 12, this can be done simultaneously for all nodes in $O(\tau_G \cdot \log n)$ rounds. At the end of the random walks, we say a token succeeds if it ends at a compartment without any other tokens. If a token does not succeed, it will go back to the origin. The process is repeated until there is no compartment containing more than one token. Since there are at most $n \cdot k \le 1.5m$ tokens, at most $1.5m$ compartment can be occupied. Since we run the random walks for $\tau_G(0.1/2m)$ steps, the probability that a random walk ends at a specific compartment is at most $1.1/2m$. Thus, the probability that a token does not succeed is at most $(1.5m) \cdot (1.1/(2m)) = 1.65/2$.

Therefore, a token will succeed w.h.p. after at most $O(\log n)$ trials. By a union bound over the tokens, w.h.p. all tokens succeed after $O(\log n)$ trials. The total running time is $O(\log n \cdot (\tau_G \log n)) = \tilde{O}(\tau_G)$.

Step 2. Each node u creates a source token. The tokens start to perform random walk for $\tau_G(\lambda')$ steps, where $\lambda' = \min(\lambda/(8m), 0.1/m)$ (see Figure 1). If the source token of u ends up in one out of the k compartments with a destination token of v, $t(u)$ will be set to v. Otherwise, if it ends up in a compartment without any destination tokens, it will restart the random walk. The process will be repeated until the source token ends up in a compartment with some destination token.

By our choice of λ', the probability that a token ends at a specific node is at least $0.9/(2m)$. Therefore, the probability that a token successfully ends up in a compartment with a destination token after the random walk is at least

$$nk \cdot \frac{0.9}{2m} \geq (1.5m - n) \cdot \frac{0.9}{2m} \geq (1.5m - m - 1) \cdot \frac{0.9}{2m} \geq \left(\frac{1}{4} - \frac{1}{2m}\right) \cdot 0.9 \geq 0.9/8 \ .$$

The second inequality follows from $m \geq n - 1$ and the third inequality holds for $m \geq 4$. Thus, the number of random walks a token needs to perform until it ends up at a node with some destination token is at most $O(\log n)$ w.h.p. By taking a union bound over all the n tokens, we conclude that w.h.p. every token performs at most $O(\log n)$ random walks. The random walks can be performed simultaneously in $O(\tau_G \cdot \log n)$ rounds, so w.h.p. the total number of rounds is $O(\tau_G \cdot \log^2 n)$.

Next, we show that given two nodes u, v, $\Pr(t(u) = v) \in [(1-\lambda)/n, (1+\lambda)/n]$. Let $\mathcal{E}_v$ denote the event that the source token of u ends up in a compartment with a destination token of v. Let $\mathcal{E}$ denote the event that the source token of u ends up in a compartment with some destination token.

By our choice of $\tau(\lambda')$, we have that for all v, $\Pr(\mathcal{E}_v) \in \left[\frac{k}{2m} - k\lambda', \frac{k}{2m} + k\lambda'\right]$ and $\Pr(\mathcal{E}) \in \left[n\left(\frac{k}{2m} - k\lambda'\right), n\left(\frac{k}{2m} + k\lambda'\right)\right]$. Therefore,

$$\begin{aligned}
\frac{\frac{k}{2m} - k\lambda'}{n\left(\frac{k}{2m} + k\lambda'\right)} &\leq \Pr(t(u) = v) \leq \frac{\frac{k}{2m} + k\lambda'}{n\left(\frac{k}{2m} - k\lambda'\right)} \\
\frac{1}{n} \cdot \frac{1 - 2m\lambda'}{1 + 2m\lambda'} &\leq \Pr(t(u) = v) \leq \frac{1}{n} \cdot \frac{1 + 2m\lambda'}{1 - 2m\lambda'} \\
\frac{1}{n} \cdot (1 - 8m\lambda') &\leq \Pr(t(u) = v) \leq \frac{1}{n} \cdot (1 + 8m\lambda') && \text{when } \lambda' \text{ is sufficiently small} \\
\frac{1}{n} \cdot (1 - \lambda) &\leq \Pr(t(u) = v) \leq \frac{1}{n} \cdot (1 + \lambda) && \lambda' \leq \lambda/8m \ .
\end{aligned}$$

Note that since all the source tokens perform random walks independently, when we condition on the choice of nodes in Z for any $u \notin Z \subseteq V$, it is still true that

$$\Pr\left(\mathcal{E}_v | \bigwedge_{z \in Z} t(z)\right) \in \left[\frac{k}{2m} - k\lambda', \frac{k}{2m} + k\lambda'\right]$$

and

$$\Pr\left(\mathcal{E} | \bigwedge_{z \in Z} t(z)\right) \in \left[n(\frac{k}{2m} - k\lambda'), n(\frac{k}{2m} + k\lambda')\right] \ .$$

Thus, $\Pr\left(t(u) = v | \bigwedge_{z \in Z} t(z)\right) \in [(1-\lambda)/n, (1+\lambda)/n]$.

Step 3. It remains to show that the messages from u to $t(u)$ can be routed simultaneously for every u in $\tilde{O}(\tau_G)$ rounds.

Let $mid(u)$ denote the node where the source token of u is located at the end of Step 2. The message from u to $mid(u)$ for every u can be simultaneously routed in $\tilde{O}(\tau_G)$ rounds by following the same path taken by the random walk of the source token of u.

Suppose that $t(u) = v$. After the message reaches $mid(u)$, it will follow the path taken by the random walk of the destination token of v to go to v (see Figure 2). Note that multiple source tokens may be matched to a node v (some possibly from the other destination tokens of v). When they follow the paths that lead back to $t(v)$, it is possible that these paths merge and create congestion. However, using a standard Chernoff Bound argument, we can show that for any node v w.h.p. at most $O(\log n)$ different nodes u have $t(u) = v$. Therefore, each step of the parallel random walk can be done with a $O(\log n)$ factor blowup. Thus, the messages between u and $t(u)$ can be routed in $\tilde{O}(\tau_G)$ rounds. This completes the proof of Theorem 4.

4 Algorithms in the GOSSIP Model

In this section, we show that if we have a small number of non-empty nodes, then the empty nodes help approximate F_p faster. As stated in Corollary 6, this result can be translated back to the CONGEST model using Theorem 4 with a blow-up factor $\tilde{O}(\tau_G)$. We exhibit a pre-processing step that duplicates the values so that $\Omega(n)$ nodes become non-empty which is crucial for the algorithms to work while preserving the occurrence ratios.

Throughout this section, for the sake of clarity, we consider the GOSSIP (0) model. However, running our algorithms in GOSSIP $(1/n^c)$, for some sufficiently large constant c, only incurs a small additive error $1/\operatorname{poly}(n)$.

▶ **Lemma 13.** *If the number of non-empty nodes $z < n/3$, we can duplicate the values so that $z\lceil (n/3)/z \rceil$ nodes become non-empty while preserving the occurrences ratios in $O(\log^2 n)$ rounds in the* GOSSIP *model.*

Proof. We divide the process into three phases.

Pre-processing. We assume that the number of non-empty nodes is less than $n/3$, otherwise, we are done. First, the nodes compute the number of non-empty nodes z in $O(\log n)$ rounds [32]. Each node v will form a token that contains $\operatorname{val}(v)$ and t where t is originally set to $\lceil (n/3)/z \rceil$.

Splitting Phase. This phase consists of $O(\log n)$ stages each of which consists of $O(\log n)$ sub-stages. At the beginning of each stage, a node v has a collection of tokens (x_1, t_1), $(x_2, t_2), \ldots$ in its buffer. Each token (x_i, t_i) is split into two tokens $(x_i, \lceil t_i/2 \rceil)$ and $(x, \lfloor t_i/2 \rfloor)$. It will send these two tokens to two random nodes using two rounds and delete (x_i, t_i) from its buffer. Note that the new tokens $(x_i, \lceil t_i/2 \rceil)$ and $(x, \lfloor t_i/2 \rfloor)$ will not be split until the next stage. Every stage produces at most $z\lceil (n/3)/z \rceil \leq 2n/3$ new tokens. Each new token is sent to a random node and therefore each node contains $O(\log n)$ new tokens w.h.p by Chernoff bound at the end of that stage. Hence, each sub-stage requires at most $O(\log n)$ rounds to split all the tokens in its buffer w.h.p. After $O(\log n)$ stages, w.h.p all nodes contain $O(\log n)$ tokens and all tokens (x, t) satisfy $t = 1$.

Distributing Phase. At this point, we only have tokens in the form $(x, 1)$, or simply x. In each stage, if v holds more than one token, it will send all but one token (say the first that arrives at v) to the nodes that it talks to. By a standard Chernoff bound argument, each stage requires $O(\log n)$ rounds since each node always holds at most $O(\log n)$ tokens

w.h.p. We say a token x succeeds if it lands in a previously empty node u while no other token lands in u in the same round. Then, u never sends x away from this point onward. Since we have at most $z \cdot \lceil (n/3)/z \rceil \leq 2n/3$ tokens, at least $n/3$ nodes are empty at all times. Consider a token x. In each stage, conditioning on all other tokens' choices, with probability at least 1/3, x succeeds. Hence, after $O(\log n)$ stages, x succeeds w.h.p and therefore all tokens succeed w.h.p by taking a union bound over all tokens. Since we have at least $\lceil n/3 \rceil$ tokens, the number of non-empty nodes is $\Omega(n)$. Note that the occurrence of each value is rescaled by a factor $\lceil (n/3)/z \rceil$. ◀

After we estimate F_p of the new instance, we can divide the estimator by $(\lceil (n/3)/z \rceil)^p$ to get an estimate for F_p in the original instance. From now on, we can safely assume that the number of non-empty nodes $F_1 = \Omega(n)$, otherwise, we can apply the above pre-processing. A *key observation* is that $F_0 \leq z$, and thus we can analyze our algorithms for when F_0 is small instead.

An ℓ_p-sampling primitive. An ℓ_p-sampling algorithm samples a value $i \in [N]$ with probability f_i^p/F_p. More formally, $\Pr(\text{sample } i) = f_i^p/F_p$.

The ℓ_p-sampling primitive (for $0 \leq p \leq 2$) has been extensively studied in the data stream model. An incomplete list includes [3, 26, 28, 36]. However, most streaming ℓ_p-samplers are rather complicated, and it is unclear how to implement them in the GOSSIP model.

It is trivial to obtain an ℓ_1-sample by virtue of the GOSSIP model. To obtain an ℓ_0-sample (a random value that occurs at least once), we broadcast a randomly chosen pairwise hash function $h : [N] \to [N^3]$ and identify the value corresponds to the smallest hash value in $O(\log n)$ rounds.

Assuming that p is fixed, we now show that if $F_0 = O\left(n^{1/(p-1)}\right)$, then we can perform ℓ_p-sampling in $O(\log n)$ rounds (hence ℓ_2-sampling can always be done in $O(\log n)$ rounds since $F_0 \leq n$). The sampling algorithm proceeds as follows.

Each node v uses p rounds to talk to p random nodes $u_1, \ldots, u_p$. It declares success if $\text{val}(u_1) = \ldots = \text{val}(u_p)$. In that case, let $\text{val}(u_1)$ be v's sample. Among the successful nodes, to break symmetry, broadcast the sample of the node with the smallest ID. If no node succeeds, repeat the process. The following lemma provides a lower bound on F_p based on F_0.

▶ **Lemma 14.** *If $F_1 = \Omega(n)$ and $F_0 = O\left(n^{1/(p-1)}\right)$, then $F_p = \Omega\left(n^{p-1}\right)$.*

Proof. Let the frequency vector be $f = (f_1, \ldots, f_N)$. Without loss of generality, suppose the potentially non-zero entries of f be $f_1, \ldots, f_{Kn^{1/(p-1)}}$ for some constant K. Note that based on our assumption, $f_j = 0$ for all $j > Kn^{1/(p-1)}$. Let $f' = (f_1, \ldots, f_{Kn^{1/(p-1)}})$ be the vector formed by the first $Kn^{1/(p-1)}$ entries. Note that $\|f'\|_1 \geq Cn$ for some constant $0 < C \leq 1$ as assumed.

We will use the following inequality: if the vector x has n entries then

$$\|x\|_q \leq \left(n^{1/q-1/p}\right) \|x\|_p \text{ , for } 0 < q < p \text{ .}$$

Note that f' has $Kn^{1/(p-1)}$ entries. Let $K' = K^{1-1/p}$. We have

$$\begin{aligned}
\left(Kn^{1/(p-1)}\right)^{1-1/p} \|f'\|_p &\geq \|f'\|_1 \\
\|f'\|_p &\geq \frac{\|f'\|_1}{K'n^{1/p}} \\
F_p &\geq \frac{C^p n^p}{K^{p-1} n} = \Omega\left(n^{p-1}\right) \text{ .}
\end{aligned}$$

The last step follows since K and C are constants and p is fixed. ◀

▶ **Theorem 15.** *If $F_0 = O\left(n^{1/(p-1)}\right)$, then the described algorithm obtains an ℓ_p-sample in $O(\log n)$ rounds in the* GOSSIP *model w.h.p.*

Proof. We can apply the pre-processing step so that $F_1 = \Omega(n)$ while the occurrences ratios are preserved. The probability that a node succeeds is $\Omega\left(\sum_{i=1}^{N} f_i^p/n^p\right) = \Omega\left(F_p/n^p\right)$.

Appealing to Lemma 14, $F_p \geq n^{p-1}/K'$ for some constant K'. Hence, $\Pr\left(v \text{ succeeds}\right) \geq 1/(K'n)$. The probability that all n nodes fail is at most $\left(1 - 1/(K'n)\right)^n \leq e^{-1/K'}$. We therefore succeed w.h.p by repeating $O(\log n)$ times. Given that v succeeds, the probability that it samples value i is $(f_i^p/n^p) / \left(\sum_{j=1}^{N} f_j^p/n^p\right) = f_i^p/F_p$ as required. ◀

Approximating F_p. The algorithm by Bar-Yossef et al. [4] that we discuss in the full version [39] for approximating F_0 up to a $1 \pm \epsilon$ factor w.h.p can be emulated in the GOSSIP model in $O(\epsilon^{-2}\log^2 n)$ rounds. We now focus on approximating higher frequency moments. Let $k \leq p$ be an integer. We present an algorithm that w.h.p approximates F_p (for $p \geq 2$) in $\tilde{O}\left(\epsilon^{-2} n^{1-k/p}\right)$ rounds if $F_0 = O\left(n^{1/(k-1)}\right)$. Recall that F_0 is at most the number of non-empty nodes. To approximate F_p, our algorithm makes use of an approximation of F_k and ℓ_k-sampling. This generalizes the approach in [3, 36]. We will prove the following theorem.

We first consider the following algorithm that approximates F_k. For $j = 1, \ldots, C\epsilon^{-2}\log n$, where C is some sufficiently large constant, in the j-th phase, each non-empty node v uses $k-1$ rounds to talk to $k-1$ random nodes $u_1, \ldots, u_{k-1}$. It declares success if $\mathrm{val}(v) = \mathrm{val}(u_1) = \ldots = \mathrm{val}(u_{k-1})$. Let $I_{j,v}$ be the indicator variable for the event v succeeds in the j-th phase. Let $T = C\epsilon^{-2}\log n$. Return the estimate

$$\hat{F}_k = \frac{n^{k-1}}{T} \cdot \sum_{j=1}^{T}\sum_{v=1}^{n} I_{j,v} \ .$$

We now prove Theorem 5. This theorem first shows that $\hat{F}_k$ is a good approximation w.h.p. Then, it combines $\hat{F}_k$ with ℓ_k-sampling to compute a good estimate of F_p in $O\left(\epsilon^{-2} n^{1-k/p}\log^2 n\right)$ rounds.

Proof of Theorem 5. We again can assume that $F_1 = z = \Omega(n)$ as outlined earlier in this section. We first show that $\hat{F}_k = (1 \pm \epsilon)F_k$ w.h.p. In expectation,

$$\mathrm{E}\left[\hat{F}_k\right] = \frac{n^{k-1}}{T}\sum_{j=1}^{T}\sum_{v=1}^{n}\mathrm{E}\left[I_{j,v}\right] = \frac{n^{k-1}}{T}\sum_{j=1}^{T}\sum_{v=1}^{n}\frac{f_{\mathrm{val}(v)}^{k-1}}{n^{k-1}} = \sum_{i=1}^{N} f_i \cdot f_i^{k-1} = F_k \ .$$

Since the indicator variables $I_{j,v}$ are independent, we can apply Chernoff bound directly.

$$\Pr\left(\left|\hat{F}_k - F_k\right| \geq \epsilon F_k\right) = \exp\left(-\Omega\left(\frac{T\epsilon^2 F_k}{n^{k-1}}\right)\right) \leq \exp\left(-\Omega\left(T\epsilon^2\right)\right) \leq 1/\mathrm{poly}(n) \ .$$

The first inequality follows from Lemma 14 and the second inequality is because $T = C\epsilon^{-2}\log n$ for some sufficiently large constant C. Hence, we can approximate F_k up to a $1 \pm \epsilon$ factor in $O(\epsilon^{-2}\log n)$ rounds.

To approximate F_p for $p > k$, we use the following estimator. Let i be an ℓ_k sample. We can compute f_i exactly in $O(\log n)$ rounds. Specifically, each node with value i will put 1 on it and 0 otherwise. Then, we can compute the sum using the algorithm in [32]. Consider the following estimator:

$$\hat{F}_p = \hat{F}_k \cdot f_i^{p-k} \ .$$

We rely on the following lemma. We defer the proof to the end of this section.

▶ **Lemma 16.** *We have* $\mathrm{E}\left[\hat{F}_p\right] = (1 \pm O(\epsilon))F_p$ *and* $\mathrm{V}\left[\hat{F}_p\right] \leq 2n^{1-k/p}F_p^2$.

Hence, by an application of Chebyshev bound, if we take the average of $O\left(n^{1-k/p}\epsilon^{-2}\right)$ estimators, with constant probability, $\hat{F}_p = (1 \pm \epsilon)F_p$. We can amplify the success probability to $1 - 1/\operatorname{poly}(n)$ by the standard median trick, i.e., taking the median of $O(\log n)$ such estimators. ◀

Proof of Lemma 16. In expectation,

$$\begin{aligned}\mathrm{E}\left[\hat{F}_p\right] &= \hat{F}_k \cdot \sum_{i=1}^{N} \frac{f_i^k}{F_k} f_i^{p-k} \\ &= (1 \pm O(\epsilon))F_p \ .\end{aligned}$$

We can bound the variance as follows.

$$\begin{aligned}\mathrm{V}\left[\hat{F}_p\right] &\leq (1 \pm O(\epsilon))F_k^2 \sum_{i=1}^{N} \frac{f_i^k}{F_k} f_i^{2(p-k)} \\ &\leq 2F_k F_{2p-k} \ .\end{aligned}$$

We have $\|f\|_k \leq n^{1/k-1/p}\|f\|_p$, and therefore $F_k \leq n^{1-k/p}F_p^{k/p}$. Additionally, $\|f\|_{2p-k} \leq \|f\|_p$ which implies $F_{2p-k} \leq F_p^{2-k/p}$. Therefore, $\mathrm{V}\left[\hat{F}_p\right] \leq 2n^{1-k/p}F_p^2$. ◀

References

1. Miklós Ajtai, János Komlós, and Endre Szemerédi. An O(n log n) Sorting Network. In *STOC*, pages 1–9. ACM, 1983.
2. Noga Alon, Yossi Matias, and Mario Szegedy. The Space Complexity of Approximating the Frequency Moments. *J. Comput. Syst. Sci.*, 58(1):137–147, 1999.
3. Alexandr Andoni, Robert Krauthgamer, and Krzysztof Onak. Streaming Algorithms via Precision Sampling. In *FOCS*, pages 363–372. IEEE Computer Society, 2011.
4. Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, D. Sivakumar, and Luca Trevisan. Counting Distinct Elements in a Data Stream. In *RANDOM*, volume 2483 of *Lecture Notes in Computer Science*, pages 1–10. Springer, 2002.
5. Vladimir Braverman and Stephen R. Chestnut. Universal Sketches for the Frequency Negative Moments and Other Decreasing Streaming Sums. In *APPROX-RANDOM*, volume 40 of *LIPIcs*, pages 591–605. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.
6. Vladimir Braverman, Stephen R. Chestnut, David P. Woodruff, and Lin F. Yang. Streaming Space Complexity of Nearly All Functions of One Variable on Frequency Vectors. In *PODS*, pages 261–276. ACM, 2016.
7. Vladimir Braverman and Rafail Ostrovsky. Zero-one frequency laws. In *STOC*, pages 281–290. ACM, 2010.
8. Joshua Brody and Amit Chakrabarti. A Multi-Round Communication Lower Bound for Gap Hamming and Some Consequences. In *IEEE Conference on Computational Complexity*, pages 358–368. IEEE Computer Society, 2009.
9. Amit Chakrabarti, Khanh Do Ba, and S. Muthukrishnan. Estimating Entropy and Entropy Norm on Data Streams. In *STACS*, volume 3884 of *Lecture Notes in Computer Science*, pages 196–205. Springer, 2006.
10. Amit Chakrabarti, Graham Cormode, and Andrew McGregor. A near-optimal algorithm for estimating the entropy of a stream. *ACM Trans. Algorithms*, 6(3):51:1–51:21, 2010.
11. Jen-Yeu Chen and Gopal Pandurangan. Almost-Optimal Gossip-Based Aggregate Computation. *SIAM Journal on Computing*, 41(3):455–483, 2012.

12 Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic Algorithms for Replicated Database Maintenance. In *Proc. 6th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 1–12, 1987.

13 A.M. Frieze and G.R. Grimmett. The shortest-path problem for graphs with random arc-lengths. *Discrete Applied Mathematics*, 10(1):57–77, 1985.

14 Sumit Ganguly. Taylor Polynomial Estimator for Estimating Frequency Moments. In *ICALP (1)*, volume 9134 of *Lecture Notes in Computer Science*, pages 542–553. Springer, 2015.

15 Mohsen Ghaffari, Fabian Kuhn, and Hsin-Hao Su. Distributed MST and Routing in Almost Mixing Time. In *PODC*, pages 131–140. ACM, 2017.

16 Mohsen Ghaffari and Jason Li. New Distributed Algorithms in Almost Mixing Time via Transformations from Parallel Algorithms. In *DISC*, volume 121 of *LIPIcs*, pages 31:1–31:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.

17 George Giakkoupis, Anne-Marie Kermarrec, and Philipp Woelfel. Gossip Protocols for Renaming and Sorting. In Yehuda Afek, editor, *DISC*, 2013.

18 Phillip B. Gibbons and Srikanta Tirthapura. Estimating simple functions on the union of data streams. In *SPAA*, pages 281–291. ACM, 2001.

19 Oded Goldreich. Basic Facts about Expander Graphs. In *Studies in Complexity and Cryptography*, volume 6650 of *Lecture Notes in Computer Science*, pages 451–464. Springer, 2011.

20 Yu Gu, Andrew McCallum, and Donald F. Towsley. Detecting Anomalies in Network Traffic Using Maximum Entropy Estimation. In *Internet Measurment Conference*, pages 345–350. USENIX Association, 2005.

21 Bernhard Haeupler, Jeet Mohapatra, and Hsin-Hao Su. Optimal Gossip Algorithms for Exact and Approximate Quantile Computations. In *PODC*, pages 179–188. ACM, 2018.

22 Nicholas J. A. Harvey, Jelani Nelson, and Krzysztof Onak. Sketching and Streaming Entropy via Approximation Theory. In *FOCS*, pages 489–498. IEEE Computer Society, 2008.

23 Shlomo Hoory, Nathan Linial, and Avi Wigderson. Expander graphs and their applications. *Bulletin of the American Mathematical Society*, 43(4):439–561, 2006.

24 Piotr Indyk. Stable distributions, pseudorandom generators, embeddings, and data stream computation. *J. ACM*, 53(3):307–323, 2006.

25 Piotr Indyk and David P. Woodruff. Optimal approximations of the frequency moments of data streams. In *STOC*, pages 202–208. ACM, 2005.

26 Rajesh Jayaram and David P. Woodruff. Perfect Lp Sampling in a Data Stream. In *FOCS*, pages 544–555. IEEE Computer Society, 2018.

27 Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. Gossip-based Peer Sampling. *ACM Trans. Comput. Syst.*, 25(3), August 2007. doi:10.1145/1275517.1275520.

28 Hossein Jowhari, Mert Saglam, and Gábor Tardos. Tight bounds for Lp samplers, finding duplicates in streams, and related problems. In *PODS*, pages 49–58. ACM, 2011.

29 Daniel M. Kane, Jelani Nelson, and David P. Woodruff. An optimal algorithm for the distinct elements problem. In *PODS*, pages 41–52. ACM, 2010.

30 R. Karp, C. Schindelhauer, S. Shenker, and B. Vocking. Randomized rumor spreading. In *Proc. 41st IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 565–574, 2000.

31 Srinivas Kashyap, Supratim Deb, K. V. M. Naidu, Rajeev Rastogi, and Anand Srinivasan. Efficient Gossip-based Aggregate Computation. In *Proc. of the 25th ACM Symposium on Principles of Database Systems (PODS)*, pages 308–317, 2006.

32 David Kempe, Alin Dobra, and Johannes Gehrke. Gossip-based computation of aggregate information. In *Proc. of the Symp. on Found. of Comp. Sci. (FOCS)*, pages 482–491, 2003.

33 Fabian Kuhn, Thomas Locher, and Stefan Schmid. Distributed computation of the mode. In *PODC*, pages 15–24. ACM, 2008.

34 Fabian Kuhn, Thomas Locher, and Rogert Wattenhofer. Tight Bounds for Distributed Selection. In *Proc. of 19th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 145–153, 2007.

35 Fabian Kuhn and Rotem Oshman. The Complexity of Data Aggregation in Directed Networks. In *DISC*, volume 6950 of *Lecture Notes in Computer Science*, pages 416–431. Springer, 2011.

36 Morteza Monemizadeh and David P. Woodruff. 1-Pass Relative-Error L_p-Sampling with Applications. In *SODA*, pages 1143–1160. SIAM, 2010.

37 David Peleg. *Distributed Computing: A Locality-sensitive Approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

38 Boris Pittel. On Spreading a Rumor. *SIAM Journal on Applied Mathematics*, 47(1):213–223, 1987.

39 Hsin-Hao Su and Hoa T. Vu. Distributed Data Summarization in Well-Connected Networks. *CoRR*, abs/1908.00236, 2019. `arXiv:1908.00236`.

40 Arno Wagner and Bernhard Plattner. Entropy Based Worm and Anomaly Detection in Fast IP Networks. In *WETICE*, pages 172–177. IEEE Computer Society, 2005.

41 David P. Woodruff. Optimal space lower bounds for all frequency moments. In *SODA*, pages 167–175. SIAM, 2004.

42 Kuai Xu, Zhi-Li Zhang, and Supratik Bhattacharyya. Profiling internet backbone traffic: behavior models and applications. In *SIGCOMM*, pages 169–180. ACM, 2005.

Polynomial-Time Fence Insertion for Structured Programs

Mohammad Taheri
Sharif University of Technology, Tehran, Iran
sm.taheri@sharif.edu

Arash Pourdamghani
Sharif University of Technology, Tehran, Iran
pourdamghani@ce.sharif.edu

Mohsen Lesani
University of California at Riverside, CA, USA
lesani@ucr.edu

Abstract

To enhance performance, common processors feature relaxed memory models that reorder instructions. However, the correctness of concurrent programs is often dependent on the preservation of the program order of certain instructions. Thus, the instruction set architectures offer memory fences. Using fences is a subtle task with performance and correctness implications: using too few can compromise correctness and using too many can hinder performance. Thus, fence insertion algorithms that given the required program orders can automatically find the optimum fencing can enhance the ease of programming, reliability, and performance of concurrent programs. In this paper, we consider the class of programs with structured branch and loop statements and present a greedy and polynomial-time optimum fence insertion algorithm. The algorithm incrementally reduces fence insertion for a control-flow graph to fence insertion for a set of paths. In addition, we show that the minimum fence insertion problem with multiple types of fence instructions is NP-hard even for straight-line programs.

2012 ACM Subject Classification Software and its engineering → Memory management; Software and its engineering → Synchronization; Software and its engineering → Concurrent programming structures

Keywords and phrases Fence Insertion, Synchronization, Concurrent Programming

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.34

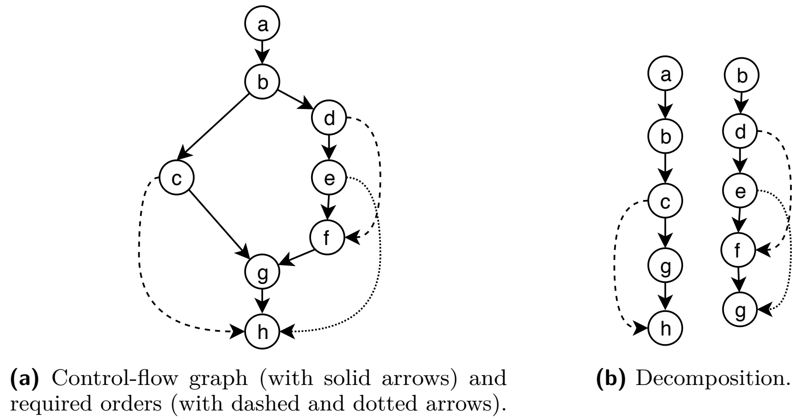
Funding This project was supported by the NSF grant #1657204.

Acknowledgements We appreciate the reviewers of our DISC 2019 submission for constructive comments.

1 Introduction

To gain performance, processors reorder instructions. However, the correctness of concurrent programs is often crucially dependent on the preservation of the order of specific instructions. For example, a flag should be set before another flag is read. Thus, architectures provide memory fence instructions that preserve the relative order of specific instructions that come before and after them in the program. Experts have been traditionally programming synchronization algorithms with explicit fence instructions for specific architectures [7, 8, 12, 14]. The resulting program is an over-specification as it hard-codes the placement of the fences that enforce the required orders for a particular architecture. Further, it is an under-specification as the required orders that are implicitly provided by the architecture do not explicitly appear in the program. Fences are just an implementation mechanism for

© Mohammad Taheri, Arash Pourdamghani, and Mohsen Lesani;
licensed under Creative Commons License CC-BY



(a) Control-flow graph (with solid arrows) and required orders (with dashed and dotted arrows).

(b) Decomposition.

Figure 1 Fence insertion for structured programs.

high-level order requirements. Concurrent algorithm designers require the order of certain instructions in their algorithms to be kept during the execution, and can readily declare these orders. Given the high-level order requirements [4, 11, 9], automatic synthesis tools that can decide the optimum fence insertion can enhance the ease of programming, reliability, maintainability, portability and performance of synchronization algorithms. This approach separates what from how. Programs can be verified using architecture-independent and algorithm-level reasoning and tools can automatically translate the program to multiple target architectures.

Lee et al. [21], Fang et al. [15], and Alglave et al. [5] presented methods to insert fences that enforce sequential consistency. Others have tried to infer the required orders including Kuperstein et al. [18], Meshman et al. [25] and Dan et al. [13]. Bender et. al. [9] proposed to capture the required orders as a relation on the statements of each thread and implemented a compiler to translate these declared orders to optimum fence insertion. Optimum fence insertion for general programs can be modeled as the minimum multi-cut problem that is NP-hard. The compiler presented in [9] used an exponential-time algorithm to insert optimum fences. Later Lesani [22] presented a polynomial-time fence insertion algorithm for the class of straight-line programs. This posed the problem of whether there are optimum or approximation algorithms for fence insertion to programs with basic control structures.

In this paper, we introduce a greedy and polynomial-time optimum fence insertion algorithm for the class of structured programs. This class includes programs that use structured branching statements such as `if-then-else` and `switch-case` and structured loop statements such as `while` and `for`. Further, through a reduction from the minimum set cover problem, we show that the minimum fence insertion problem with multiple types of fence instructions is NP-hard even for straight-line programs.

We observed that the control-flow graph of programs with structured branching has the structured form of nested diamonds. Figure 1.(a) shows an example. The diamond that branches at vertex b and merges at vertex g can represent an if-then-else statement where the left branch represents the then statement and the right branch represents the else statement. The example requires three orders: the order from c to h, from d to f, and from e to h should be enforced. The orders are shown as arrows. We use both dashed and dotted arrows to easily distinguish overlapping orders. The order between two vertices can be preserved in a path between them by placing a fence on an edge of the path. What is the minimum number of fences to preserve all the required orders?

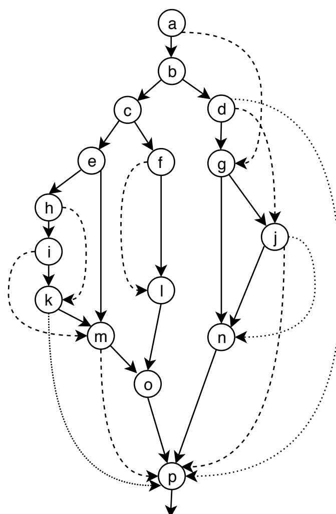


Figure 2 An example of AFG and its constraints. A constraint $\langle s, t\rangle$ is shown as a dashed arrow from the source s to the sink t. The diamonds $\langle e, m\rangle$ and $\langle g, n\rangle$ are at level 0. The diamond $\langle c, o\rangle$ is at level 1 and the diamond $\langle b, p\rangle$ is at level 2. The constraint $\langle h, k\rangle$ is an internal constraint for the diamond $\langle e, m\rangle$, the constraint $\langle d, j\rangle$ is a spanning constraint for the diamond $\langle g, n\rangle$, and The constraint $\langle d, p\rangle$ is a passing constraint for the diamond $\langle g, n\rangle$.

In this paper, we present an algorithm that reduces fence insertion for structured control-flow-graphs to fence insertion for a set of paths. It also presents a transformation that reduces fence insertion for looping structured programs to loop-free structured programs. For example, fence insertion for the graph shown in Figure 1.(a) is reduced to fence insertion for the two paths shown in Figure 1.(b). The high-level idea is that we can incrementally transform a diamond to a single branch by extracting branches. Fences can be independently inserted for the extracted branches. For example, the right branch of Figure 1.(a) is extracted in Figure 1.(b). The orders within a branch can be only preserved by fences inserted within that branch. Thus, the extracted right branch takes the order from d to f with it. Further, the extracted right branch can cover the spanning order from e to h with no extra fences. Thus, it takes in the spanning order from e to h too; it takes it as the shrunk order from e to g. Thus, fence insertion for the extracted right branch covers both constraints from d to f and from e to h. The left branch and the vertices above and below the diamond make the second path. The order from c to h overlaps with the left branch and stays within the second extracted path. The result is two paths and fencing for each can be done in polynomial time. We will elaborate on the algorithm in the following sections.

In the following sections, we first define the problem model (Section 2) and then present the greedy fence insertion algorithm for loop-free structured programs and state its optimality and complexity (Section 3). We then present a reduction from fence insertion for looping programs to fence insertion for loop-free programs (Section 4). Then, we prove the NP-hardness of fence insertion with multiple fence types (Section 5). Finally, we discuss the related works (Section 6) before we conclude (Section 7).

2 Problem Model

We now present the basic definitions and the problem instances that we use throughout the paper.

We consider the problem of *minimum fence insertion* for the set of *structured programs* that use branching statements such as if-then-else and switch-case and loop statement such as while and for. We represent this problem as the pair $\langle G, C\rangle$. The graph $G = \langle V, E\rangle$ is the control-flow graph (CFG) of the program. Each vertex $v \in V$ represents an executable instruction, and each edge $e \in E$ represents an execution transition. The control-flow graphs for loop-free structured programs are acyclic; thus, we call them *Acyclic Flow Graphs (AFG)*. The *constraints* C is the set of pairs of vertices of G that represent the required orders: a constraint is represented as a pair $\langle s, t\rangle$, where s and t are both vertices in V such that t is reachable from s. Figure 2 illustrates an instance of the fence insertion problem. The order between two vertices of a constraint can be preserved in a path between them by placing a fence on an edge of the path. To preserve the required order of a constraint between two vertices, a fence should be inserted in each path between them; then, we say that the constraint is *covered* by the inserted fences. Common hardware memory models do not allow reordering at the branch instructions. Therefore, we assume that branch instructions have implicit fences. Given a problem instance $\langle G, C\rangle$, the goal is to find the minimum number of fences that preserve all the constraints. We call the set of fences inserted on the edges of a graph, a *fencing* for that graph.

An AFG has a structured shape of nested diamonds. It has only one vertex with the input degree 0 that we call the start vertex and denote by v_0. It has only one vertex with the output degree 0 that we call the end vertex. Vertices with output degree more than one are called *branch* vertices and denoted by b. Vertices with input degree more than one are called *merge* vertices and denoted by m. The branch vertices start and the merge vertices end diamonds. Diamonds will help us reduce the problem on large graphs into a set of simple paths. A *simple path* is a path that has no branch vertex or merge vertex, except its head and tail. Diamonds are nested. A pair of $\langle b,m\rangle$ is a diamond of level 0 (called a *simple diamond*) iff (1) b is a branch vertex and m is a merge vertex and (2) all the paths starting from b reach m and all such paths are simple. In Figure 2, $\langle e, m\rangle$ and $\langle g, n\rangle$ are diamonds of level 0. A pair of $\langle b, m\rangle$ is a diamond of level k iff (1) b is a branch vertex and m is a merge vertex and (2) all the paths starting from b reach m, and if there is any branch vertex in between, it should be the starting vertex of a diamond of a level less than k whose merge vertex is not m. In Figure 2, $\langle c, o\rangle$ is a diamond of level 1 and $\langle b, p\rangle$ is a diamond of level 2.

Given a diamond, we categorize constraints into three types with respect to that diamond. A constraint is *internal* if both end points of the constraint are in the diamond. For example, in Figure 2, the constraint $\langle h, k\rangle$ is an internal constraint for the diamond $\langle e, m\rangle$. A constraint is *spanning* if only one of its endpoints is in the diamond. For example, in Figure 2, the constraints $\langle d, j\rangle$ and $\langle j, p\rangle$ are spanning constraints for the diamond $\langle g, n\rangle$. A constraint is *passing* if it passes over the diamond. More precisely, the branch vertex of the diamond is reachable from the source vertex of the constraint, the sink vertex of the constraint is reachable from the merge vertex of the diamond and every path from the source to the sink vertex of the constraint contains branch and merge vertices of the diamond. For example, in Figure 2, the constraint $\langle d, p\rangle$ is a passing constraint for the diamond $\langle g, n\rangle$. Since the branch vertex of a diamond is a jump instruction and the end vertex of a diamond represents the end label of the branch, there is no constraint between the two in real-world programs. Thus, we assume that constraints do not start at the branch vertex and finish at the merge vertex of a diamond.

3 Fence Insertion Algorithm for Loop-free Programs

Algorithm 1 Fence Insertion.

1: **procedure** FenceInsertion ($\langle AFG, C\rangle$)
2: Eliminate the constraints in C that are implicitly preserved in AFG.
3: Find diamonds in AFG and
4: store them in a minimum priority queue q based on their levels. (Algorithm 2)
5: Decompose the diamonds in q into a set of simple paths and their constraints and
6: find the optimum fencing for each. (Algorithm 3 and Algorithm 4)
7: Return the union of the fences placed on the simple paths.

In this section, we present an algorithm (Algorithm 1) that finds the optimal solution to the fence insertion problem for a given AFG in polynomial time. The algorithm has three steps. In the first step, it eliminates the constraints that are implicitly preserved, from the AFG. In the second step, it finds the diamonds of the AFG and puts them into a minimum priority queue based on their levels. The idea behind the algorithm for this step is to run a breadth-first search to label the merge and branch vertices, and then match them up accordingly. These labels are also used to assign the level of each diamond. The third step of the algorithm iterates through the diamonds in the priority queue and decomposes them into a set of separate simple paths. In this step, it also calculates the optimal fencing for each of the simple paths. Finally, it returns the union of these fencings. We will visit each of these steps in turn.

3.1 Constraint Elimination

Hardware memory models such as x86-TSO, SPARC TSO, MIPS and RISC-V can preserve control dependencies [27, 6, 3]. Therefore, spanning constraints that start before the branch vertex of a diamond and end inside a branch path of the diamond are implicitly preserved. Similarly, the passing constraints are implicitly preserved. For example, in the graph of Figure 2, the spanning constraints $\langle a, g\rangle$ and $\langle d, j\rangle$, and the passing constraint $\langle d, p\rangle$ are implicitly preserved. As Figure 5.(a) shows, the implicitly provided constraints are eliminated from the graph before the next steps. These constraints can be simply eliminated as follows: traverse the vertices from the start vertex, and at each branch vertex, eliminate the constraints which have been started but not finished.

3.2 Finding diamonds

In this section, we present an algorithm (Algorithm 2) that finds all the diamonds of the input graph and computes their levels. It returns a minimum priority queue of the diamonds based on their levels.

The algorithm starts from the start vertex v_0 of the graph and traverses the vertices by a breadth-first-search. It calculates a label for each vertex. It initializes the label of the start vertex v_0 to 0. In each iteration, a vertex v is visited. If v is not a branch vertex, the algorithm sets the label of v to the maximum of the labels of its parents. On the other hand, the labels advance on each branch. If v is a branch vertex, it sets the label of v to one plus the maximum of the labels of its parents. It also adds the branch vertex to a stack. If v is a merge vertex, its corresponding branch vertex b is popped from the stack. A new diamond is found with the branch vertex b and merge vertex v. The level of the diamond is the difference of the labels of the merge and branch vertices. When a diamond is found, it is added to a priority queue based on its level. Finally, the priority queue is returned.

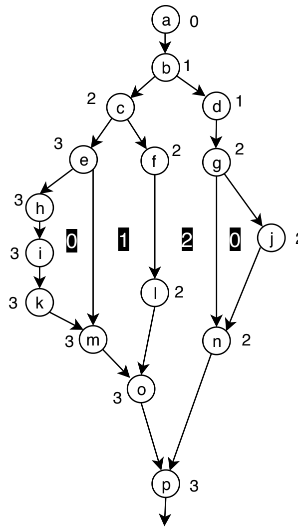
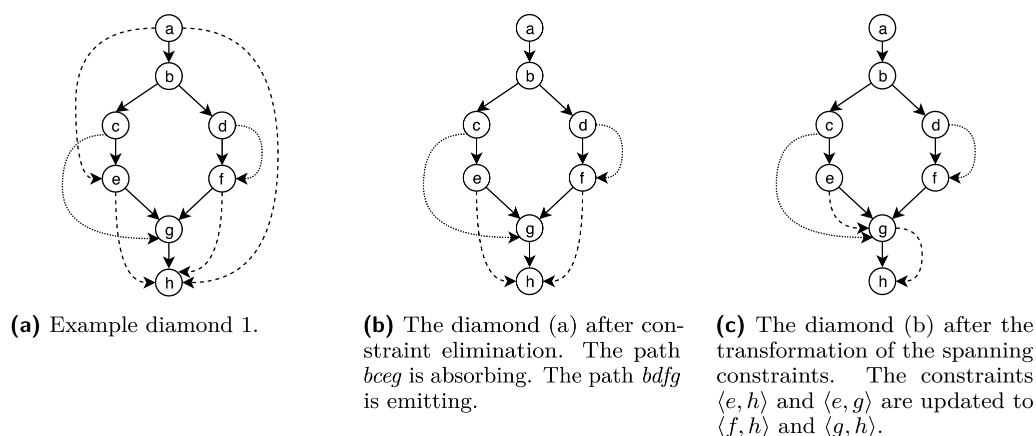


Figure 3 An example execution of Algorithm 2 that finds diamonds on the graph in Figure 2. The calculated label of each vertex is shown close to it. Four diamonds are found. The numbers with the dark background show the level of the enclosing diamonds. $\mathit{level}(\langle e, m\rangle) = 0$, $\mathit{level}(\langle g, n\rangle) = 0$, $\mathit{level}(\langle c, o\rangle) = 1$, $\mathit{level}(\langle b, p\rangle) = 2$.

As an example, Figure 3 shows the execution of Algorithm 2 on the graph in Figure 2. The calculated label of each vertex is shown close to it and the numbers with the dark background show the level of the enclosing diamonds. The label of the start vertex a is 0. The labels of the subsequent branch vertices b, c and e are 1, 2 and 3 respectively. The labels of the merge vertices m and o are both 3. Thus, the level of the diamond $\langle e, m\rangle$ is 0 and the level of the diamond $\langle c, o\rangle$ is 1. The algorithm finds four diamonds with the following levels: $\mathit{level}(\langle e, m\rangle) = 0$, $\mathit{level}(\langle g, n\rangle) = 0$, $\mathit{level}(\langle c, o\rangle) = 1$, and $\mathit{level}(\langle b, p\rangle) = 2$.

Algorithm 2 Finding Diamonds

1: **procedure** FINDDIAMONDS (AFG)
2: ▷ $\mathit{parents}(v)$ and $\mathit{children}(v)$ return parents and children of v in AFG
3: ▷ Uses the FIFO queue q, the marked set m, the branch stack s and the priority queue p
4: $\mathit{label}(v_0) := 0$
5: Add v_0 to q
6: **while** q is not empty **do**
7: Pop v from q
8: Add v from m
9: $\mathit{label}(v) := \max_{p \in \mathit{parents}(v)} \mathit{label}(p)$
10: **if** v is a branch vertex **then**
11: Push v to s
12: $\mathit{label}(v) := \mathit{label}(v) + 1$
13: **if** v is a merge vertex and all of its parents are in m **then**
14: Pop branch vertex b from the stack
15: Add diamond $\langle b, v\rangle$ with level $\mathit{label}(v) - \mathit{label}(b)$ to p
16: **else**
17: continue.
18: Add $\mathit{children}(v)$ that are not in m to q
19: return p



(a) Example diamond 1.

(b) The diamond (a) after constraint elimination. The path *bceg* is absorbing. The path *bdfg* is emitting.

(c) The diamond (b) after the transformation of the spanning constraints. The constraints $\langle e, h\rangle$ and $\langle e, g\rangle$ are updated to $\langle f, h\rangle$ and $\langle g, h\rangle$.

Figure 4 Transformation of spanning constraints.

3.3 Decomposing Diamonds into Simple Paths

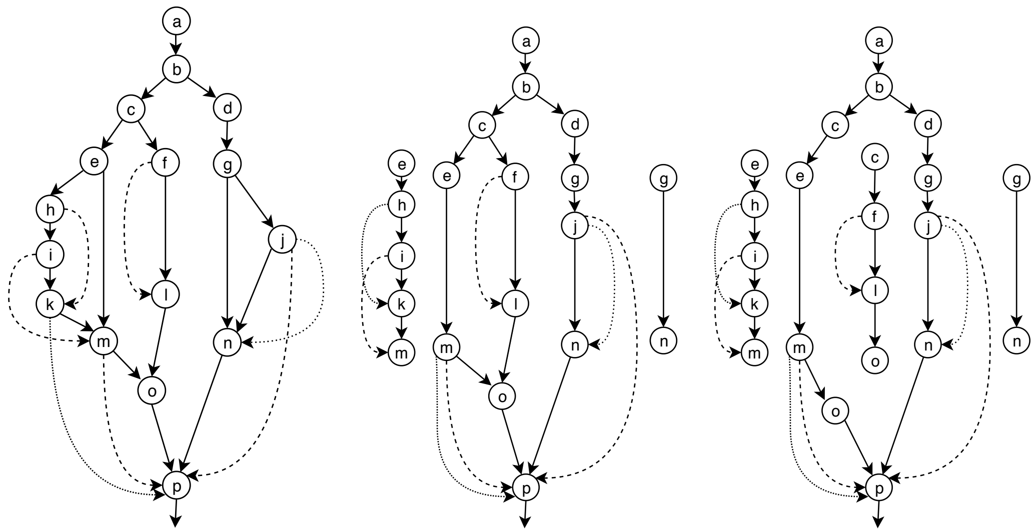
Algorithm 3 Decomposing Diamonds into a Set of Simple Paths.

```
1:  procedure FenceInsertion (q)  ▷ q is the minimum priority queue ordering diamonds by level
2:      Initialize the set F to ∅.
3:      while q is not empty do
4:          Extract the innermost diamond d from q.
5:          while there is more than one path in d do
6:              Pick a path p in d.
7:              Call Algorithm 4 on p to find the fencing f and the type t.
8:              Add f to F
9:              if t is absorbing then
10:                 Update the end point of the spanning constraints to the merge point of d.
11:             else                                                        ▷ t is emitting
12:                 Update the start point of the spanning constraints to the merge point of d.
13:             Remove p from d.
14:     return F
```

In this step, we present an algorithm (Algorithm 3) that decomposes each diamond into simple paths and finds the optimum fencing for them. The algorithm iterates the diamonds from the innermost to the outermost. For each diamond, it incrementally extracts simple paths until only a simple path remains in the diamond. Therefore, the degree of the nesting diamond decreases from one to zero. This makes the nesting diamond a simple diamond. As the algorithm iterates all the nested diamonds before the nesting one, diamonds are visited when they are already simple.

For each path of a diamond, the algorithm calls the fence insertion algorithm for simple paths (that we will see in Algorithm 4) to obtain an optimum fence placement for the internal constraints of the path. The rationale for the separation of paths is that the internal constraints of a path can be covered by only fences inside the path. Thus, the optimum fencing for the internal constraints can be locally determined. The algorithm then checks if the resulting fence placement can cover the spanning constraints of the path.

Accordingly there are two path types: absorbing and emitting. We use an example in Figure 4 to illustrate these types. Figure 4.(a) shows a simple diamond. Figure 4.(b) shows the resulting diamond after eliminating constraints that are implicitly preserved by the control dependencies. We illustrate the two path types on the diamond in Figure 4.(b).



(a) The graph of Figure 2 after constraint elimination.

(b) Decomposing diamonds of level 0.

(c) Decomposing diamonds of level 1.

Figure 5 Decomposition of Nested Diamonds into a Set of Simple Paths.

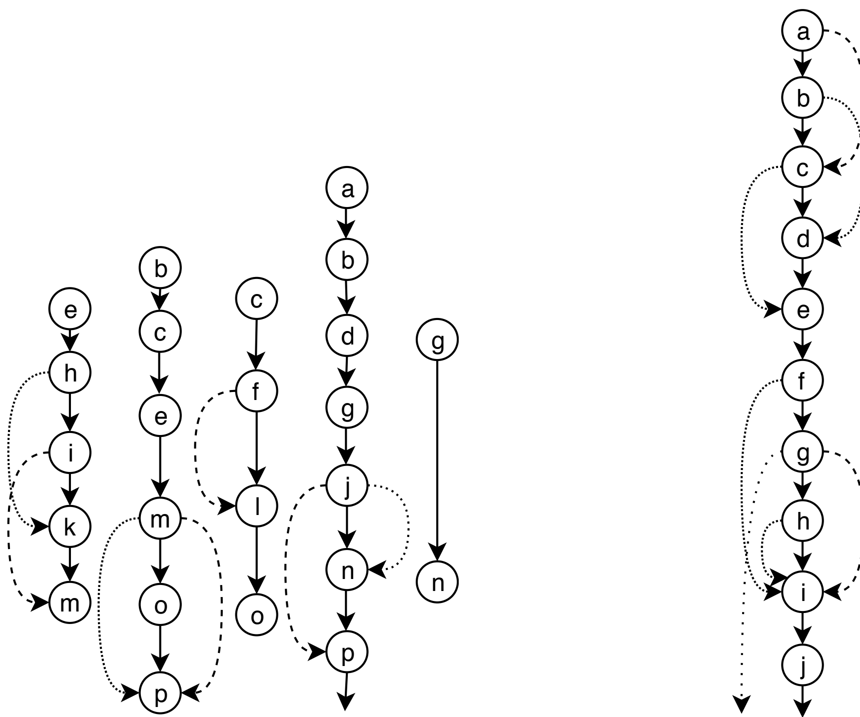


Figure 6 The Final Set of Simple Paths for Figure 5.(a)

Figure 7 Fence insertion for a simple path. The algorithm visits the constraints in the order $\langle a, c\rangle$, $\langle b, d\rangle$, $\langle c, e\rangle$, $\langle f, i\rangle$, $\langle g, i\rangle$, and $\langle h, i\rangle$ and inserts the fences $\langle b, c\rangle$, $\langle d, e\rangle$, and $\langle h, i\rangle$. The inserted fences cover the spanning constraint starting from g.

- Absorbing: A path of a simple diamond is absorbing if the required fences for its internal constraints can cover its spanning constraints as well. For example, the paths *bceg* in Figure 4.(b) is absorbing because a fence at $\langle e, g \rangle$ handles both constraints.
- Emitting: A path of a simple diamond is emitting if it is not absorbing. In other words, a path is emitting if the required fences for its internal constraints cannot cover its spanning constraints. For example, the paths *bdfg* in Figure 4.(b) is emitting. The optimum fencing for the path *bdfg* in Figure 4.(b) is one fence on the edge $\langle d, f \rangle$ that does not cover the constraint $\langle f, h \rangle$.

To extract simple paths from a diamond, its spanning constraints should be transformed to be totally in or out of the path. The algorithm updates the spanning constraints of the paths according to their types. Absorbing paths absorb them inside and emitting paths emit them outside of the path. The rationale behind this transformation is that an absorbing path can cover the spanning constraint with no extra fence in the path. Thus, extra constraints are covered without increasing the number of fences. Therefore, the spanning constraint is pulled inside the path. On the other hand, in an emitting path, an extra fence is needed to cover the spanning constraint. This extra fence cannot cover any additional constraints inside the path but may cover other constraints outside the path. Therefore, the spanning constraint is pushed outside. Thus, the algorithm performs the following two transformations. (1) Transformation for absorbing paths: The spanning constraints stay in the path. The endpoints of the spanning constraints are updated to the merge point of the diamond. For example, in Figure 4.(b), the constraint $\langle e, h \rangle$ is updated to $\langle e, g \rangle$. (2) Transformation for emitting paths: The spanning constraints are pushed out of the path. The start points of the spanning constraints are updated to the merge point of the diamond. For example, in Figure 4.(b), the constraint $\langle f, h \rangle$ is updated to $\langle g, h \rangle$. We note that the transformation leaves the internal and passing constraints unchanged.

We illustrate the iteration over diamonds of different levels in Figure 5. Constraint elimination on the graph in Figure 2 results in Figure 5.(a). Figure 5.(b) shows the result of Algorithm 3 on Figure 5.(a) after processing the diamonds of level 0. The diamonds of level 0 are $\langle e, m \rangle$ and $\langle g, n \rangle$. For the diamond $\langle e, m \rangle$, the left simple path *ehikm* is extracted. The constraint $\langle k, p \rangle$ is a spanning constraint for this path. The optimum fencing for the internal constraints of this path is one fence on the edge $\langle i, k \rangle$ that does not cover the spanning constraint $\langle k, p \rangle$. So the path is emitting and the constraint $\langle k, p \rangle$ is shrunk to $\langle m, p \rangle$. Extracting the left path reduces the diamond to a simple path. The other diamond of level 0 is $\langle g, n \rangle$. The left edge $\langle g, n \rangle$ with no constraint can be simply extracted to reduce the diamond to a simple path.

Figure 5.(c) shows the result of Algorithm 3 on the graph of Figure 5.(b) after processing the diamonds of the next level, that has been level 1 in the original graph of Figure 5.(a). The only diamond of the next level is $\langle c, o \rangle$. There are no spanning constraints and simply extracting the right path *cflo* reduces the diamond to a simple path. Figure 5.(c) has only one diamond $\langle b, p \rangle$ left. After Algorithm 3 processes this diamond, the end result is the set of simple paths shown in Figure 6. The diamond $\langle b, p \rangle$ has no spanning constraints and it is simply split into two simple paths. The graph is decomposed into five separate simple paths.

3.4 Fence Insertion for Simple Paths

In this section, we present an algorithm (Algorithm 4) that finds the optimum fencing for simple paths. More precisely, given the internal constraints of a simple path and a bottom spanning constraint, the algorithm finds a minimal fence placement that covers the internal constraints and also decides whether the spanning constraint can be covered by no more fences. The algorithm can be trivially extended for more spanning constraints.

Algorithm 4 Fence Insertion and Deciding the Type for a Simple Path.

```
1: procedure FenceInsertionForSimplePath (C, s)
2:                ▷ C is the set of internal constraints and s is the spanning constraint.
3:     Initialize F to ∅.
4:     Sort C to a list L according to the end point in the top-to-bottom order.
5:     for (each constraint c in L in order) do
6:         Add to F a fence f on the last edge of c.
7:         Remove from C the constraints that are covered by f.
8:     if (f covers s) then
9:         Return ⟨F, absorbing⟩
10:    else
11:        Return ⟨F, emitting⟩
```

We illustrate the algorithm using the simple path shown in Figure 7 as an example. The example path has the set C of six internal constraints and a spanning constraint s at the bottom with the start vertex g and no end vertex. The algorithm first sorts the given set C of internal constraints to a list L according to their endpoints in the top-to-bottom order. In the example, the sorted order can be $\langle a, c\rangle$, $\langle b, d\rangle$, $\langle c, e\rangle$, $\langle f, i\rangle$, $\langle g, i\rangle$, and $\langle h, i\rangle$. It then iterates over constraints in L in order. For the current constraint c, the algorithm adds a fence at the bottom edge of c. It then removes any later constraint that is covered by the inserted fence. The rationale for putting the fence at the bottom edge is to cover the current constraint and also reach as far down as possible to cover the later constraints if possible. In the example, first the constraint $\langle a, c\rangle$ is visited and a fence is inserted at the edge $\langle b, c\rangle$. This fence covers the constraint $\langle b, d\rangle$ as well; so, it is removed from L. Next, the constraint $\langle c, e\rangle$ is visited that results in the insertion of the fence $\langle d, e\rangle$. Similarly, the next constraint $\langle f, i\rangle$ results in the fence $\langle h, i\rangle$. This fence covers the other two constraints as well. So the resulting set F of fences is $\langle b, c\rangle$, $\langle d, e\rangle$, and $\langle h, i\rangle$. The algorithm then checks whether the inserted fences cover the spanning constraint as well. If it does, the path is absorbing; otherwise, the path is emitting. In this example, the spanning constraint starting from g is covered by the inserted fence $\langle h, i\rangle$. So, the algorithm returns the set of fences F and that the path is absorbing.

3.5 Optimality and Complexity

In this section, we show that the optimality of the algorithm. We show that every optimal solution needs at least the number of fences that the algorithm inserts. In addition, we show the time and space complexity of the algorithm.

▶ **Theorem 1.** *Algorithm 4 provides optimum fence insertion for simple paths.*

Proof. The proof is by the following pair of facts. First, the size of the optimum solution is at least the size of every set of non-overlapping constraints. Second, the constraints that lead to addition of fences are non-overlapping. The algorithm visits the constraints by their endpoints, inserts fenced in the last edges of constraints, and removes all the covered constraints. Thus, if a fence is inserted for a constraint, its start point can be only at or after the end point of the last constraint that required a fence; thus, the two constraints do not overlap. ◀

▶ **Theorem 2.** *Algorithm 1 provides optimal fence insertion for AFGs.*

Proof. We prove this fact by induction on the level of diamonds. In the base case, suppose we have a diamond of level 0. The internal constraints of a branch can be covered by only fences inside the branch. Algorithm 1 uses Algorithm 4 to find the fencing for the branches of the diamond. By Theorem 1, each of these fencings are optimal for the internal constraints of that branch. Further, Algorithm 4 puts fences on the lowest possible edges. At the end, it checks whether the inserted fences can cover the spanning constraint as well and accordingly decides whether the paths are of absorbing or emitting type. Based on the type of the path, Algorithm 1 transforms the spanning constraints of the path: an absorbing path absorbs it inside and an emitting path emits it outside of the path. An absorbing path can cover the spanning constraint with no extra fence in the path. Thus, extra constraints are covered without increasing the number of fences. Therefore, the solution stays optimum after pulling the spanning constraint inside the path. On the other hand, in an emitting path, an extra fence is needed to cover the spanning constraint. If an extra fence is inserted inside the path, it cannot cover any additional constraints inside or outside the path. However, if it is put outside the path, it may cover other overlapping constraints. Therefore, pushing the spanning constraint outside can result in either the same or fewer number of fences. In the inductive case, consider a diamond of level k. Algorithm 1 reduces nested diamonds of lower levels to simple paths; thus, the diamond is reduced to a diamond of level 0. With the same argument as the base case, Algorithm 1 finds the optimum fencing. ◀

▶ **Theorem 3.** *Algorithm 1 is of* $\mathcal{O}(|C|log|C| + |C||V| + |V|log|V|)$ *time and* $\mathcal{O}(|C| + |V|)$ *space complexity.*

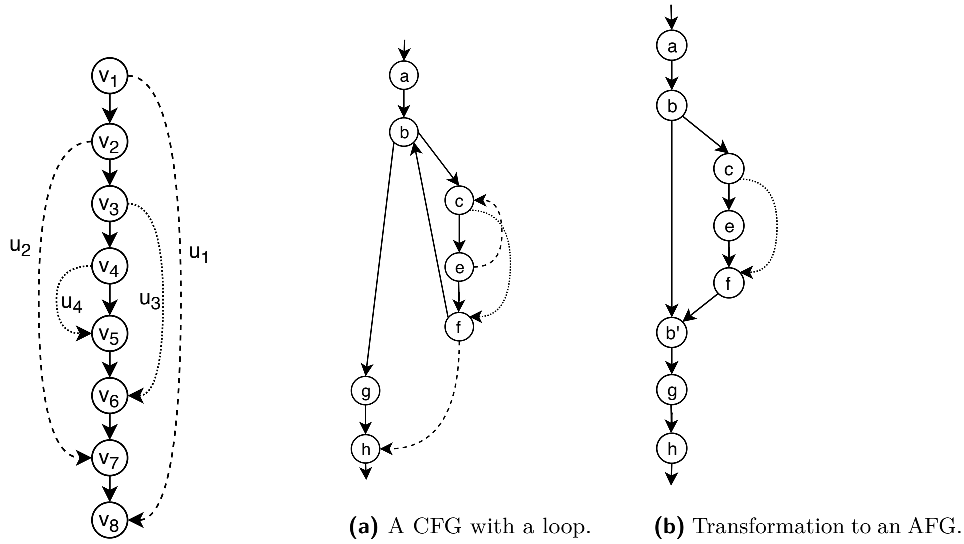
Proof. Algorithm 1 has three steps and its time complexity is the sum of their complexity. We consider each step in turn. We note that it takes $\mathcal{O}(|C|+|E|)$ space to represent the input.

In the first step, we eliminate the constraints that are implicitly covered by the branch vertices. The algorithm traverses the vertices from the start vertex, and for each branch vertex eliminates the constraints which have been started but not finished. Since each edge and each constraint is visited just once, the running time is $\mathcal{O}(|C| + |E|)$. Additionally, we do not need any extra memory for running this step. Therefore, its space complexity is $\mathcal{O}(|C| + |E|)$.

In the second step, the algorithm finds the diamonds (Algorithm 2) by traversing the graph vertices using a breath-first-search and pushes the diamonds into a priority queue. So, it takes $\mathcal{O}(|E|log|E| + |V|)$ time. The algorithm uses data structures that store vertices and thus, needs $\mathcal{O}(|V|)$ extra space for this step.

In the third step, the algorithm decomposes the diamonds (Algorithm 3) into simple paths and finds the optimum fencing for them. Algorithm 3 applies Algorithm 4 to each branch of each diamond. For a path p, let C_p be set of constraints on p. Algorithm 4 sorts C_p, which takes $\mathcal{O}(|C_p|log|C_p|)$ and then traverses C_p and inserts fences which takes $\mathcal{O}(|C_p|)$. Therefore, its time complexity is $\mathcal{O}(|C|log|C|)$. Also, it needs at most $\mathcal{O}(|E|)$ space to represent the fencing. Thus, fence insertion for the branches takes $\mathcal{O}(|C|log|C|)$ time and $\mathcal{O}(|E|)$ space for all the diamonds. In addition, Algorithm 3 updates spanning constraints for branches and extracts branches of each diamond. The graph has at most $\mathcal{O}(|E|)$ diamonds and in the worst case, a constraint may need to be updated when each diamond is visited. Therefore, updating the constraints takes $\mathcal{O}(|E||C|)$ time. It only needs $\mathcal{O}(|E|)$ additional space to represent the extracted paths.

We now sum the complexity of the steps. The time complexity of Algorithm 1 is $\mathcal{O}(|V| + |E|log|E| + |C|\log|C| + |C||E|)$. The space complexity of Algorithm 1 is $\mathcal{O}(|C| + |E| + |V|)$. To further simplify these orders, we show that $|E| \in \mathcal{O}(|V|)$. It is easy to see that the



(a) A CFG with a loop. (b) Transformation to an AFG.

Figure 8 Reduction Example.

Figure 9 Converting a Loop to a Diamond.

sum of the degree of all the merge vertices of an AFG is $\mathcal{O}(|V|)$. Similarly, the sum of the degrees of all the branch vertices of an AFG is $\mathcal{O}(|V|)$. Also, the sum of the degrees of all non-merge non-branch vertices is $\mathcal{O}(|V|)$. As a result, the sum of degrees of all the vertices is $\mathcal{O}(|V|)$ thus, $|E| \in \mathcal{O}(|V|)$. Therefore, the time complexity of Algorithm 1 is $\mathcal{O}(|V|log|V| + |C|log|C| + |C||V|)$ and its space complexity is $\mathcal{O}(|C| + |V|)$ ◀

4 Fence Insertion for Loops

In this section, we present a transformation for loops in a given CFG with loops to an AFG. Therefore, we can reduce fence insertion for any CFG to an AFG and use Algorithm 1 to find an optimal fence insertion.

We illustrate the transformation using an example. Figure 9.(a) shows a CFG with a loop. The vertex b is the branch instruction: it jumps either to the body of loop at the vertex c or out of the loop to vertex g. We call the edge $\langle b, c\rangle$ that jumps from the branch vertex to the loop body, the start edge. The body of the loop is a CFG in general. In this example, it is the simple path cef. We call the edge $\langle f, b\rangle$ that jumps from the end of the loop body back to the branch vertex, the return edge. We call the edge $\langle b, g\rangle$ that jumps from the branch vertex out of the loop, the exit edge.

We now transform the CFG in Figure 9.(a) to the AFG in Figure 9.(b). The graph has two internal constraints in the loop body: $\langle e, c\rangle$ and $\langle c, f\rangle$, and the constraint $\langle f, h\rangle$ from inside the loop body to outside of the loop. The constraint $\langle e, c\rangle$ is upwards: it requires the execution of e in one iteration of the loop to be executed before the execution of c in the next iteration of the loop. We notice that the branch instruction b is executed between the instructions of any iteration and the next. As mentioned in Subsection 3.1, hardware memory models can preserve control dependencies. Thus, the order of instruction between one iteration and the next is preserved. Therefore, the constraint $\langle e, c\rangle$ in Figure 9.(a) is implicitly enforced and is eliminated in Figure 9.(b). The constraint $\langle f, h\rangle$ will be eliminated with the same argument. Thus, we should preserve the constrains when the loop body is either executed once in an iteration or is not executed. To represent these two paths in a

diamond, we add a vertex b' to represent a dummy instruction after the loop. The return edge $\langle f, b\rangle$ is updated to $\langle f, b'\rangle$. The exit edge $\langle b, g\rangle$ is updated to a dummy edge $\langle b, b'\rangle$ and the edge $\langle b', g\rangle$. Thus, the loop is transformed to the diamond $\langle b, b'\rangle$ and the constraint $\langle c, f\rangle$ remain unchanged.

No constraint ends at b'. Thus, Algorithm 4 puts no fence on the dummy edge $\langle b, b'\rangle$. In addition, the transformation did not change the constraints in the body or out of the body of the loop. Any fence on the new edge $\langle b', g\rangle$ corresponds to a fence on the old edge $\langle b, g\rangle$. Therefore, if a fence is needed in the resulting AFG, it is needed in the CFG as well and will cover the same set of constraints. Therefore, every optimal fence insertion for the AFG is an optimal fence insertion for the CFG.

5 Multi-type Fence Insertion Problem

Common architectures often offer different fence instructions that preserve the order of certain instruction pairs. In this section, we study the complexity of the insertion problem when there are different types of constraints and fences such that each fence type can cover a subset of the constraint types. (We note that these constraint types are defined based on the endpoint instructions for a target architecture, and are irrelevant to the three constraint types presented in Section 2.) We show that this problem is NP-hard even for straight-line programs through a reduction from the set cover problem.

An instance of the *Multi-type Fence Insertion* problem is defined as $\langle CT, FT, G, C\rangle$. CT is the set of constraint types. Each constraint has a type according to its endpoint instructions. FT is the set of fence types. Each fence type can cover a certain subset of the constraints types CT. G is the CFG. C is the set of constraints on G of different types from CT. The goal is to find the minimum number of fences, regardless of their types from FT, to cover C.

We provide a polynomial-time reduction from the minimum set cover problem to the multi-type fence insertion problem. The set cover has been one of the fundamental problems in computer science [17]. It has been shown that the minimum set cover problem is NP-hard [10] and it can be approximated with a $O(log\ n)$ factor [16].

▶ **Theorem 4.** *The multi-type fence insertion problem is NP-hard.*

Proof. We provide a reduction from an instance I of the minimum set cover problem to an instance of I' the multi-type fence insertion problem for straight-line programs. Consider an instance I of the minimum set cover problem $\langle U, S\rangle$ where $U = \{u_1, u_2, \ldots, u_n\}$ is the set of all elements and $S = \{S_1, S_2, S_3, \ldots, S_k\}$ are the subsets of U. The goal is to find the minimum number of the subsets in S that cover U.

The reduction defines the set of constraint types CT to be U. Each element of the set U corresponds to constraint type. The reduction also defines the set of fence types FT to be S. Each fence type S_i covers the set of constraint types that correspond to the elements in S_i. The reduction constructs a straight-line program with $2n$ instructions. Then, for each element $u_i \in U$, it creates a constraint $c_i = \langle v_i, v_{2n-i}\rangle$ with type u_i. Therefore, each constraint c_i will starts at the vertex v_i and ends at the vertex v_{2n-i} and can be covered by a fence S_i that includes the element u_i. Let us call the constructed instance I'.

As an example consider a minimum set cover instance with $U = \{u_1, u_2, u_3, u_4\}$ and $S = \{\{u_1, u_3\}, \{u_1, u_2, u_4\}, \{u_3, u_2\}\}$. As Figure 8 shows, it is reduced to a straight-line program with 8 instructions. The set of constraint types is U and the set of fence types is S. The constraints will be $\langle v_1, v_8\rangle$, $\langle v_2, v_7\rangle$, $\langle v_3, v_6\rangle$ and $\langle v_4, v_5\rangle$ of types u_1, u_2, u_3 and u_4 respectively.

First, we show that given a solution of the set cover instance I, a solution for the multi-type fence insertion problem I' can be constructed. Consider that the solution of I has chosen the subsets S_i to cover U. In the solution of I', we use the fences corresponding to the subsets S_i that I has chosen. The fences are all put in the middle edge of G. Since the subsets S_i cover all the elements u in U, the inserted fences S_i can cover the constraints that are of any type u in U. All the constructed constraints are of a type u in U; thus all of them are covered by the inserted fences. Next, we show that given a solution for the minimum multi-type fence insertion I', a solution for the minimum set cover I can be constructed. Any solution for the multi-type fence insertion can be transformed to a solution for it with the same number and type of fences by moving the fences to the middle edge. This is because the middle edge is in the middle of all the constraints. Thus, we consider an optimum solution for I' that has all the fences in the middle edge. A solution for the minimum set cover I can be constructed by simply choosing the sets S_i that correspond to the inserted fences in the solution of I'. Because otherwise, a smaller set cover for I can be transformed to a smaller fence insertion for I' that contradicts the assumption that the solution for I' is optimum. ◀

6 Related Work

In this paper, we introduced a polynomial-time algorithm to find the optimal fence insertion for structured programs and showed that fence insertion with multiple fence types is NP-hard. The previous methods that involve fence insertion can be grouped into the following categories.

Sequential Consistency: There have been attempts to insert fences to enforce sequential consistency. Lee et al. [21] used delay set analysis and dominators to reduce the number of fences that provide sequential consistency. To preserve sequential consistency during compilation, Fang et al. [15] applied several techniques to enhance fence insertion for memory models of specific architectures (SMPs on IBM Power 3 and Pentium 4). Linden et al. [23] presented a heuristic approach to output a correct but maybe suboptimal fence insertion to preserve sequential consistency on the x86-TSO memory model. Abdulla et al. [1, 2] applied reachability analysis for TSO and PSO memory models to optimize fence insertion for finite-state programs. Also, Alglave et al. [5] presented a practical and approximate static approach for fence insertion. They showed that certain cycles represent the violation of sequential consistency, statically detect the cycles and insert fences to remove them. The above related works try to preserve sequential consistency, are often empirical and do not focus on optimality guarantees. We observe that the correctness of concurrent programs is often dependent on only a few crucial orders rather than complete sequential consistency. Given these required orders, this paper presented a fence insertion algorithm that finds the optimum fencing to preserve them.

Inference: A few projects applied different techniques to automatically infer the required orders for the correctness of concurrent programs. Given a program, a correctness property, and a memory model, Kuperstein et al. [20] infer the required execution orders. They perform a whole-program state-space exploration that produces a logical formula, solve the formula to get a set of execution orders, and use those orders to insert fences. This approach infers sound but maybe suboptimal fence orders. Their follow-up works [19, 25, 13] extend the approach to degrees of infinite-state programs. Liu et al. [24] presented a dynamic inference approach that tests the input program to expose violations and adds orders to prevent the violations. The fence insertion algorithm presented in this paper takes the required orders as input and finds the optimum fencing to preserve them. The tools above can assist algorithm designers to declare the set of orders that are sufficient for correctness of the program.

Fence Elimination: To reduce the number of fences on a given relaxed memory model, there are practical techniques that eliminate redundant fences. Vafeiadis et al. [28] remove redundant fences that precede later fences or locked instructions. Morisset et al. [26] remove redundant fences for x86, ARM, and Power in the LLVM backend. Unlike the two above works that present correct techniques to reduce the number of fences, this paper proposes a fence-insertion algorithm with proof of optimality.

Hardness and Optimality: Lee et al. [21] showed that the decision version of the fence insertion problem with one type of fence on a general graph is NP-Complete. Bender et al. [9] implemented an exponential algorithm to compile declared orders to the optimum fencing. Lesani [22] focused on the limited class of straight-line programs and presented a polynomial-time algorithm. We observed that CFGs of structured programs have structured forms that can make the problem solvable in polynomial-time.

7 Conclusion

This paper considered the fence insertion problem for the class of structured programs and presented a greedy and polynomial-time optimum fence insertion algorithm. The algorithm reduces fence insertion for a control-flow graph (CFG) to fence insertion for a set of paths. It transforms looping CFGs to loop-free CFGs that are a set of nested diamonds. It then iterates the diamonds from the innermost to the outermost and incrementally extracts branches. Fence insertion for the extracted paths can be done independently and in polynomial time. This paper also proved that fence insertion with multiple fence types is NP-hard even for straight-line programs through a reduction from the set cover problem.

8 Future Work

This paper poses new avenues of investigation:

Multi-fence algorithms: If we assume that the number of different fence types is a constant k, the question is whether there is a polynomial-time parametrized algorithm for k. Otherwise, as the problem is NP-hard, the only other possible option is approximation algorithms.

Stochastic optimization: This paper presented an algorithm to minimize the number of fences. However, in common executions, some paths may be exercised more often than others. Given a probabilistic measure of how often each branch is executed, the question is to find the fencing that minimizes the probabilistic number of executed fences.

References

1. Parosh Aziz Abdulla, Mohamed Faouzi Atig, Magnus Lång, and Tuan Phong Ngo. Precise and sound automatic fence insertion procedure under PSO. In *International Conference on Networked Systems*, pages 32–47. Springer, 2015.
2. Parosh Aziz Abdulla, Mohamed Faouzi Atig, and Tuan-Phong Ngo. The best of both worlds: Trading efficiency and optimality in fence insertion for TSO. In *European Symposium on Programming Languages and Systems*, pages 308–332. Springer, 2015.
3. Sarita V Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *computer*, 29(12):66–76, 1996.
4. Jade Alglave and Patrick Cousot. Ogre and Pythia: an invariance proof method for weak consistency models. In *ACM SIGPLAN NOTICES*, volume 52 (1), pages 3–18. ACM, 2017.

5 Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Don't sit on the fence: A static analysis approach to automatic fence insertion. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 39(2):6, 2017.

6 Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in weak memory models. In *International Conference on Computer Aided Verification*, pages 258–272. Springer, 2010.

7 Hagit Attiya, Danny Hendler, and Smadar Levy. An O (1)-barriers optimal RMRs mutual exclusion algorithm. In *Proceedings of the 2013 ACM symposium on Principles of distributed computing*, pages 220–229. ACM, 2013.

8 Hagit Attiya, Danny Hendler, and Philipp Woelfel. Trading fences with rmrs and separating memory models. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 173–182. ACM, 2015.

9 John Bender, Mohsen Lesani, and Jens Palsberg. Declarative Fence Insertion. In *OOPSLA*, 2015.

10 Korte Bernhard and J Vygen. Combinatorial optimization: Theory and algorithms. *Springer, Third Edition, 2005.*, 2008.

11 Karl Crary and Michael J Sullivan. A calculus for relaxed memory. In *ACM SIGPLAN Notices*, volume 50 (1), pages 623–636. ACM, 2015.

12 Luke Dalessandro, Michael F Spear, and Michael L Scott. NOrec: streamlining STM by abolishing ownership records. In *ACM Sigplan Notices*, volume 45 (5), pages 67–78. ACM, 2010.

13 Andrei Marian Dan, Yuri Meshman, Martin Vechev, and Eran Yahav. Predicate abstraction for relaxed memory models. In *International Static Analysis Symposium*, pages 84–104. Springer, 2013.

14 Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *International Symposium on Distributed Computing*, pages 194–208. Springer, 2006.

15 Xing Fang, Jaejin Lee, and Samuel P Midkiff. Automatic fence insertion for shared memory multiprocessing. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 285–294. ACM, 2003.

16 Uriel Feige. A threshold of ln n for approximating set cover. *Journal of the ACM (JACM)*, 45(4):634–652, 1998.

17 Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

18 Michael Kuperstein, Martin Vechev, and Eran Yahav. Automatic Inference of Memory Fences. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, FMCAD '10, pages 111–120, Austin, TX, 2010. FMCAD Inc.

19 Michael Kuperstein, Martin Vechev, and Eran Yahav. Partial-coherence Abstractions for Relaxed Memory Models. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 187–198, New York, NY, USA, 2011. ACM. doi:10.1145/1993498.1993521.

20 Michael Kuperstein, Martin Vechev, and Eran Yahav. Automatic inference of memory fences. *ACM SIGACT News*, 43(2):108–123, 2012.

21 Jaejin Lee and David A Padua. Hiding relaxed memory consistency with compilers. In *Parallel Architectures and Compilation Techniques, 2000. Proceedings. International Conference on*, pages 111–122. IEEE, 2000.

22 Mohsen Lesani. Brief Announcement: Fence Insertion for Straight-line Programs is in P. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 97–99. ACM, 2017.

23 Alexander Linden and Pierre Wolper. A verification-based approach to memory fence insertion in relaxed memory systems. In *International SPIN Workshop on Model Checking of Software*, pages 144–160. Springer, 2011.

24 Feng Liu, Nayden Nedev, Nedyalko Prisadnikov, Martin Vechev, and Eran Yahav. Dynamic synthesis for relaxed memory models. *ACM SIGPLAN Notices*, 47(6):429–440, 2012.

25 Yuri Meshman, Andrei Dan, Martin Vechev, and Eran Yahav. Synthesis of memory fences via refinement propagation. In *International Static Analysis Symposium*, pages 237–252. Springer, 2014.

26 Robin Morisset and Francesco Zappa-Nardelli. Partially redundant fence elimination for x86, ARM, and Power processors. In *Proceedings of the 26th International Conference on Compiler Construction*, pages 1–10. ACM, 2017.

27 Susmit Sarkar, Peter Sewell, Francesco Zappa-Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O Myreen, and Jade Alglave. The semantics of x86-CC multiprocessor machine code. *ACM SIGPLAN Notices*, 44(1):379–391, 2009.

28 Viktor Vafeiadis and Francesco Zappa-Nardelli. Verifying fence elimination optimisations. In *International Static Analysis Symposium*, pages 146–162. Springer, 2011.

Brief Announcement: On Self-Adjusting Skip List Networks

Chen Avin
Communication Systems Engineering Department,
Ben Gurion University of the Negev, Beersheva, Israel
avin@cse.bgu.ac.il

Iosif Salem
Faculty of Computer Science, University of Vienna, Austria
iosif.salem@univie.ac.at

Stefan Schmid
Faculty of Computer Science, University of Vienna, Austria
stefan_schmid@univie.ac.at

Abstract

This paper explores the design of dynamic network topologies which adjust to the workload they serve, in an online manner. Such self-adjusting networks (SANs) are enabled by emerging optical technologies, and can be found, e.g., in datacenters. SANs can be used to reduce routing costs by moving frequently communicating nodes topologically closer. This paper presents SANs which provide, for the first time, provable *working set* guarantees: the routing cost between node pairs is proportional to how recently these nodes communicated last time. Our SANs rely on skip lists (which serve as the topology) and provide additional interesting properties such as local routing.

2012 ACM Subject Classification Networks → Topology analysis and generation

Keywords and phrases self-adjusting networks, skip lists, working set, online algorithms

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.35

1 Introduction

We design scalable and robust self-adjusting networks (SANs) [1, 5], e.g., providing not only working set guarantees but also logarithmic diameter, low degree, and connectivity even after a failure. To this end, we consider SAN topologies based on skip lists [4]. Skip lists are not only interesting for data structures but also for networks as they, e.g., provide *local routing*. This is particularly useful in the context of dynamic topologies, which change over time, since we do not have to distribute information about new routing tables.

The main **contribution** of this paper is the first SAN that achieves the *pairwise* working set property. To this end, we formally define a natural notion of working set which depends on the number of distinct nodes that participated in communication requests, since the last requests that included the corresponding source and destination nodes. Our algorithm is based on a straight-forward extension of classic self-adjusting data structures, using a "move-to-front" (MTF) data structure as a subroutine.

2 $SASL^2$: A Self-Adjusting Algorithm for Skip List Networks

We first provide some background knowledge and modeling details. Then we present our self-adjusting algorithm for skip list networks, $SASL^2$.

Skip lists networks. The skip list [4] was designed as a search data structure that serves as a probabilistic alternative to balanced trees. Let $X = \{x_1, x_2, \ldots, x_n\}$ be a set of integer keys (or items or elements) such that each x_i is associated with a node v_i. We also consider

© Chen Avin, Iosif Salem, and Stefan Schmid;
licensed under Creative Commons License CC-BY

33rd International Symposium on Distributed Computing (DISC 2019).
Editor: Jukka Suomela; Article No. 35; pp. 35:1–35:3

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Algorithm 1 $SASL^2$: Self-adjusting Algorithm for Skip List Networks.

```
1  upon request (u, v)
2    route (u, v);
3    for x ∈ {u, v} do adjustSASL(x);
4  adjustSASL(z)
5    b ← B(z);
6    promotion(z, B_1);
7    UpdCountersProm(z);
8    for i = 1, ..., b − 1
9      x ← RandomSelect(i);
10     demotion(x, B_{i+1});
11     UpdCountersDem(x);
```

two special nodes *head* and *tail*, with keys $-\infty$ and $+\infty$, respectively. Given a coin with a fixed probability of heads p, each node decides on the height of its key $h(x_i)$, by starting at height 1 and increasing the height by one for each flip that is heads until the first time the coin flip is tails. The height $H = \max_i h(x_i)$ of the skip list is expected to be in $\mathcal{O}(\log n)$.

The skip list is formed by connecting vertically H doubly-linked lists that contain subsets of $X \cup \{-\infty, +\infty\}$ linked in ascending order. We denote these lists by $\mathcal{L}_1, \dots, \mathcal{L}_H$, where $|\mathcal{L}_i| = \Theta(2^i)$ and $\mathcal{L}_i \subset \mathcal{L}_{i+1}$, for $i \in \{1, \dots, H-1\}$. All lists start and end with $-\infty$ and $+\infty$. The bottom list $\mathcal{L}_H$ contains all the keys and list $\mathcal{L}_i$ includes all items of height at least i. We assume that bideractional vertical pointers link occurrences of each node x_i in adjacent lists $\mathcal{L}_i$ and $\mathcal{L}_{i+1}$, $i = 1, \dots, H-1$. We refer to an item's right neighbor in a list as its successor and to its left neighbor as its predecessor. The search procedure for a node v starts at the top of $-\infty$ and proceeds forward until reaching a node $w \neq v$ such that its successor has a larger key than $key(v)$. Then, the search moves to the list below and continues until the same condition is satisfied. The search ends either (successfully) when the node is found, or when it reaches the bottom list and the condition for moving lower holds.

A skip list can be also viewed as a graph or skip list *network*, where the node set is $V = \{v_1, v_2, \dots, v_n\}$, denoting routers or servers, and two nodes are connected with a bidirectional link if their keys are adjacent at some level of the skip list. Each node v_i stores the triples (h, dir, x), for each level $h = 1, 2, \dots, h(x_i)$, direction $dir \in \{left, right\}$, and adjacent key $x \in \{x_1, x_2, \dots, x_n\}$. The data structure and graph point of view are equivalent.

Routing in skip list networks. In data structure terms, a routing request is quite similar to finger search [2], i.e. a search request that originates in an item resp. *node* $u \neq -\infty$ towards another node v. We consider the following procedure that requires only local information. If $u < v$ ($u > v$) then the routing proceeds to the right (left). The routing procedure is split in an up-phase and a down-phase. The routing path starts at the highest level of u with the up-phase. During the up-phase, at the current item the path moves up if the next item is smaller (larger) than v, unless the top level is reached, in which case it moves to the right (left) and repeats. When the next item is larger (smaller) than v, then the down-phase begins, which is essentially a standard skip list search for v. That is, at the current item, the path moves to the right (left) if the next item is smaller (larger) than v, otherwise it moves one level down and repeats the rightward (leftward) search, until locating v.

Prior work: Randomized self-adjusting skip lists for search sequences. Ciriani et al. [3] presented $SASL$, an online Self-Adjusting Skip List algorithm for sequences of search requests, that achieves static optimality, i.e. it performs as well as the static offline algorithm. $SASL$ is based on the following three principles: (a) logically partition the levels of a skip list $\mathcal{L}$ in a $\mathcal{O}(\log \log n)$ number of *bands* (sets of consecutive lists) of exponentially increasing size from top to bottom, (b) upon search of an element move it to the top band, and (c) if the searched element was *associated* with the band x, demote an element uniformly at random

(using a random walk) for each band $\mathcal{B}_i$ to $\mathcal{B}_{i+1}$, for $i \in [1, x-1]$. To drive the random walk, each node x maintains a set of counters $c_i(x)$ that keeps the number of elements reachable in each band $\mathcal{B}_i$ via a classical skip list search starting from x.

Extending from data structures to networks: $SASL^2$. We present an extension of $SASL$ to the case of routing in self-adjusting skip list networks. Our algorithm $SASL^2$ (Algorithm 11) uses the adjustment part of $SASL$, i.e. the promotion and demotion procedures, as a black box. Let $adjustSASL(z)$ be the adjustment part of $SASL$ with input z [3]. Upon a communication request (u, v), $SASL^2$ serves the request and then calls $adjustSASL(u)$ and subsequently $adjustSASL(v)$. A call to $adjustSASL(z)$ promotes z to the top band, $\mathcal{B}_1$ (line 6), updates the counters of a skip list search to z after its promotion (line 7), and then demotes an element x uniformly at random from $\mathcal{B}_i$ to $\mathcal{B}_{i+1}$, for $i = 1, \ldots, b-1$ (lines 8–11), where b (line 5) is the band in which z belonged before its promotion. For a single demotion from band $\mathcal{B}_i$, $adjustSASL(z)$ first selects uniformly at random a node x from band $\mathcal{B}_i$ (line 9), reduces x's height such that x belongs to $\mathcal{B}_{i+1}$ (line 10), and updates the counters in a skip list search to x before and after its demotion to the next band (line 11).

Working set theorem for sequences of communication requests in $SASL^2$. Intuitively, the working bag of a communication request $\sigma_t = (s_t, d_t)$ is the shortest suffix of the sequence $\sigma = (\sigma_1, \ldots, \sigma_{t-1})$ that includes requests in which s_t and d_t appear. The size of the working bag is the working bag number. The working set includes all distinct elements in the working bag and the working set number is the size of the working set. Our working bag and set definitions are suitable for topologies that have a front/top, and it is thus possible to design algorithms that follow a move-to-front/move-to-top principle. The motivation for these definitions is that pairs of nodes that appear in a lot of searches separately should have a relatively small joint working bag and set.

▶ **Definition 1** (Working bag and working bag number). *Let $\sigma = (\sigma_t = (s_t, d_t))_{t \in \{1,\ldots,m\}}$ be a sequence of communication requests. We define the (pairwise) working bag of a communication request $\sigma_t = (s_t, d_t)$ to be $(\sigma_1, \ldots, \sigma_t)$, if s_t or d_t appear in a request of σ for the first time at time t and $WB(s_t, d_t) = \min\{\sigma' \sqsubseteq (\sigma_1, \ldots, \sigma_{t-1}) \,|\, last(\sigma') = \sigma_{t-1} \wedge \exists_{\sigma_i, \sigma_j \in \sigma'} s_t \in \sigma_i \wedge d_t \in \sigma_j\}$ otherwise, where $\sqsubseteq$ denotes the suffix relation and $last(\sigma')$ returns the last request of a sequence σ'. We denote by $|WB(s_t, d_t)|$ the size of $WB(s_t, d_t)$ and refer to it as (s_t, d_t)'s working bag number.*

▶ **Definition 2** (Working set and working set number). *The (pairwise) working set of a communication request $\sigma_t = (s_t, d_t) \in \sigma$ is $WS(\sigma_t) = WS(s_t, d_t) = \{x \in \sigma_i \,|\, \sigma_i \in WB(s_t, d_t)\}$. The working set number of σ_t is the size of $WS(s_t, d_t)$ and we denote it by $|WS(s_t, d_t)|$.*

▶ **Theorem 3.** *For any communication request (u, v), $SASL^2$ achieves the pairwise working set property: $E[cost(SASL^2(u, v))] = \mathcal{O}(\log |WS(u, v)|)$.*

References

1. Chen Avin and Stefan Schmid. Toward demand-aware networking: a theory for self-adjusting networks. *ACM SIGCOMM Computer Communication Review*, 48(5):31–40, 2019.
2. Amitabha Bagchi, Adam L Buchsbaum, and Michael T Goodrich. Biased skip lists. *Algorithmica*, 42(1):31–48, 2005.
3. Valentina Ciriani, Paolo Ferragina, Fabrizio Luccio, and S Muthukrishnan. A data structure for a sequence of string accesses in external memory. *ACM TALG*, 3(1):6, 2007.
4. William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.
5. Stefan Schmid et al. Splaynet: Towards locally self-adjusting networks. *IEEE/ACM Transactions on Networking (TON)*, 24(3):1421–1433, 2016.

Brief Announcement: Streaming and Massively Parallel Algorithms for Edge Coloring

Soheil Behnezhad
Department of Computer Science, University of Maryland, College Park, MD, USA

Mahsa Derakhshan
Department of Computer Science, University of Maryland, College Park, MD, USA

MohammadTaghi Hajiaghayi
Department of Computer Science, University of Maryland, College Park, MD, USA

Marina Knittel
Department of Computer Science, University of Maryland, College Park, MD, USA

Hamed Saleh
Department of Computer Science, University of Maryland, College Park, MD, USA

Abstract

A valid *edge-coloring* of a graph is an assignment of "colors" to its edges such that no two incident edges receive the same color. The goal is to find a proper coloring that uses few colors. In this paper, we revisit this problem in two models of computation specific to massive graphs, the *Massively Parallel Computations* (MPC) model and the *Graph Streaming* model:

Massively Parallel Computation. We give a randomized MPC algorithm that w.h.p., returns a $(1+o(1))\Delta$ edge coloring in $O(1)$ rounds using $\widetilde{O}(n)$ space per machine and $O(m)$ total space. The space per machine can also be further improved to $n^{1-\Omega(1)}$ if $\Delta = n^{\Omega(1)}$. This is, to our knowledge, the first constant round algorithm for a natural graph problem in the strongly sublinear regime of MPC. Our algorithm improves a previous result of Harvey et al. [SPAA 2018] which required $n^{1+\Omega(1)}$ space to achieve the same result.

Graph Streaming. Since the output of edge-coloring is as large as its input, we consider a standard variant of the streaming model where the output is also reported in a streaming fashion. The main challenge is that the algorithm cannot "remember" all the reported edge colors, yet has to output a proper edge coloring using few colors.
We give a one-pass $\widetilde{O}(n)$-space streaming algorithm that always returns a valid coloring and uses 5.44Δ colors w.h.p., if the edges arrive in a random order. For adversarial order streams, we give another one-pass $\widetilde{O}(n)$-space algorithm that requires $O(\Delta^2)$ colors.

2012 ACM Subject Classification Theory of computation → Massively parallel algorithms; Theory of computation → Streaming, sublinear and near linear time algorithms

Keywords and phrases Massively Parallel Computation, Streaming, Edge Coloring

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.36

Acknowledgements Supported in part by Guggenheim Fellowship, NSF grants CCF:SPX 1822738, IIS:BIGDATA 1546108, DARPA grant SI3CMD, UMD Year of Data Science Program Grant, and Northrop Grumman Faculty Award.

1 Introduction & Results

Given a graph $G(V, E)$, an edge coloring of G is an assignment of "colors" to the edges in E such that no two incident edges receive the same color. The goal is to find an edge coloring that uses few colors. Edge coloring is among the most fundamental graph problems and has been studied in various models of computation, especially in distributed and parallel settings. In this paper, we study edge coloring in models that target massive graphs. Specifically, we focus on the *Massively Parallel Computations* (MPC) model and the *Graph Streaming* model.

© Soheil Behnezhad, Mahsa Derakhshan, MohammadTaghi Hajiaghayi, Marina Knittel, and Hamed Saleh;
licensed under Creative Commons License CC-BY

33rd International Symposium on Distributed Computing (DISC 2019).
Editor: Jukka Suomela; Article No. 36; pp. 36:1–36:3

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The MPC Model & the Related Work. The MPC model is a popular abstraction of modern parallel frameworks such as MapReduce, Hadoop, Spark, etc. We have seen a plethora of results on graph problems ever since the formalization of MPC. The studied problems include matching and vertex cover [3, 5, 7, 9], maximal independent set [9, 10], vertex coloring [4, 6, 10, 12, 13], as well as graph connectivity and related problems [1, 2, 11].

Not much work has been done on the edge coloring problem in the MPC model. The only exception is the algorithm of Harvey et al. [10] which roughly works by random partitioning the *edges*, and then coloring each partition in a different machine using a sequential $(\Delta+1)$ edge coloring algorithm. The choice of the number of partitions leads to a trade-off between the number of colors used and the space per machine required. The main shortcoming of this idea, however, is that if one desires a $\Delta + \widetilde{O}(\Delta^{1-\Omega(1)})$ edge coloring, then a strongly super linear local space of $n\Delta^{\Omega(1)}$ is required. In comparison, for the related $(\Delta+1)$ *vertex* coloring problem, Assadi et al. [4] recently presented an algorithm that takes $O(1)$ rounds and requires a near linear space of $\widetilde{O}(n)$. Unfortunately, this progress on vertex coloring does not imply a better edge coloring MPC algorithm even if we consider the more relaxed $(2\Delta-1)$ edge coloring problem. The reason is that the well-known reduction, which yields a $(2\Delta-1)$ edge coloring via a $(\Delta+1)$ vertex coloring on the line-graph, is not applicable in the MPC model as the line-graph may be significantly larger than the original graph.

Our main result is the following algorithm which achieves a near-optimal edge coloring within a constant number of rounds using a near-linear in n space.

▶ **Theorem 1.** *There exists an MPC algorithm that using $O(n)$ space per machine and $O(m)$ total space, returns a $\Delta + \widetilde{O}(\Delta^{3/4})$ edge coloring in $O(1)$ rounds.*

The Streaming Model. In the standard graph streaming model, the edges of a graph arrive one by one and the algorithm has a space that is much smaller than the total number of edges. A particularly important choice of space is $\widetilde{O}(n)$ – which is also known as the *semi-streaming* model [8] – so that the algorithm has enough space to store the vertices but not the edges. For edge coloring, the output is as large as the input, thus, we cannot hope to be able to store the output in bulk at the end. For this, we consider a standard twist on the streaming model where the output is also reported in a streaming fashion. This model is referred to in the literature as the "W-streaming" model. We particularly focus on one-pass algorithms.

Note that designing one-pass W-streaming algorithms is particularly challenging since the algorithm cannot "remember" all the choices made so far (e.g., the reported edge colors). Therefore, even the sequential greedy algorithm for $(2\Delta-1)$ edge coloring, which iterates over the edges in an arbitrary order an assigns an available to each color upon visiting it, cannot be implemented since we are not aware of the colors used incident to an edge.

Our first result presented in Theorem 2 is to show that a natural algorithm w.h.p. provides an $O(\Delta)$ edge coloring if the edges arrive in a random-order. Further, we show that for any arbitrary arrival of edges, there is a one-pass $\widetilde{O}(n)$ space W-streaming edge coloring algorithm that succeeds w.h.p. and uses $O(\Delta^2)$ colors.

▶ **Theorem 2.** *If the edges arrive in a random-order, there is a one-pass $\widetilde{O}(n)$ space W-streaming edge coloring algorithm that always returns a valid edge coloring and w.h.p. uses $(2e+o(1))\Delta \approx 5.44\Delta$ colors.*

References

1 Alexandr Andoni, Aleksandar Nikolov, Krzysztof Onak, and Grigory Yaroslavtsev. Parallel algorithms for geometric graph problems. In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 574–583, 2014. doi:10.1145/2591796.2591805.

2 Alexandr Andoni, Zhao Song, Clifford Stein, Zhengyu Wang, and Peilin Zhong. Parallel Graph Connectivity in Log Diameter Rounds. In *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018, Paris, France, October 7-9, 2018*, pages 674–685, 2018. doi:10.1109/FOCS.2018.00070.

3 Sepehr Assadi, MohammadHossein Bateni, Aaron Bernstein, Vahab S. Mirrokni, and Cliff Stein. Coresets Meet EDCS: Algorithms for Matching and Vertex Cover on Massive Graphs. *Proceedings of the 30th annual ACM-SIAM Symposium on Discrete Algorithms (SODA), to appear*, 2019.

4 Sepehr Assadi, Yu Chen, and Sanjeev Khanna. Sublinear Algorithms for $(\Delta+1)$ Vertex Coloring. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 767–786, 2019. doi:10.1137/1.9781611975482.48.

5 Soheil Behnezhad, MohammadTaghi Hajiaghayi, and David G. Harris. Exponentially Faster Massively Parallel Maximal Matching. In *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, to appear*, 2019.

6 Yi-Jun Chang, Manuela Fischer, Mohsen Ghaffari, Jara Uitto, and Yufan Zheng. The Complexity of $(\Delta + 1)$ Coloring in Congested Clique, Massively Parallel Computation, and Centralized Local Computation. *CoRR*, abs/1808.08419, 2018. arXiv:1808.08419.

7 Artur Czumaj, Jakub Lacki, Aleksander Madry, Slobodan Mitrovic, Krzysztof Onak, and Piotr Sankowski. Round Compression for Parallel Matching Algorithms. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018, Los Angeles, CA, USA, June 25-29, 2018*, pages 471–484, 2018. doi:10.1145/3188745.3188764.

8 Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. On Graph Problems in a Semi-streaming Model. In *Automata, Languages and Programming: 31st International Colloquium, ICALP 2004, Turku, Finland, July 12-16, 2004. Proceedings*, pages 531–543, 2004. doi:10.1007/978-3-540-27836-8_46.

9 Mohsen Ghaffari, Themis Gouleakis, Christian Konrad, Slobodan Mitrovic, and Ronitt Rubinfeld. Improved Massively Parallel Computation Algorithms for MIS, Matching, and Vertex Cover. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018, Egham, United Kingdom, July 23-27, 2018*, pages 129–138, 2018. doi:10.1145/3212734.3212743.

10 Nicholas J. A. Harvey, Christopher Liaw, and Paul Liu. Greedy and Local Ratio Algorithms in the MapReduce Model. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, pages 43–52, New York, NY, USA, 2018. ACM. doi:10.1145/3210377.3210386.

11 Tomasz Jurdzinski and Krzysztof Nowicki. MST in $O(1)$ rounds of congested clique. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 2620–2632, 2018. doi:10.1137/1.9781611975031.167.

12 Merav Parter. $(\Delta + 1)$ Coloring in the Congested Clique Model. In *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*, pages 160:1–160:14, 2018. doi:10.4230/LIPIcs.ICALP.2018.160.

13 Merav Parter and Hsin-Hao Su. Randomized $(\Delta + 1)$-Coloring in $O(\log^* \Delta)$ Congested Clique Rounds. In *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018*, pages 39:1–39:18, 2018. doi:10.4230/LIPIcs.DISC.2018.39.

Brief Announcement: Memory Lower Bounds for Self-Stabilization

Lélia Blin
Sorbonne Université, CNRS, Université Evry-Val d'Essonne, LIP6 UMR 7606, Paris
lelia.blin@lip6.fr

Laurent Feuilloley
Sorbonne Université, CNRS, LIP6 UMR 7606, Paris
laurent.feuilloley@lip6.fr

Gabriel Le Bouder
Sorbonne Université, CNRS, ENS Paris-Saclay, LIP6 UMR 7606, Paris
gleboude@ens-paris-saclay.fr

Abstract

In the context of self-stabilization, a *silent* algorithm guarantees that the communication registers (a.k.a register) of every node do not change once the algorithm has stabilized. At the end of the 90's, Dolev et al. [Acta Inf. '99] showed that, for finding the centers of a graph, for electing a leader, or for constructing a spanning tree, every silent deterministic algorithm must use a memory of $\Omega(\log n)$ bits per register in n-node networks. Similarly, Korman et al. [Dist. Comp. '07] proved, using the notion of proof-labeling-scheme, that, for constructing a minimum-weight spanning tree (MST), every silent algorithm must use a memory of $\Omega(\log^2 n)$ bits per register. It follows that requiring the algorithm to be silent has a cost in terms of memory space, while, in the context of self-stabilization, where every node constantly checks the states of its neighbors, the silence property can be of limited practical interest. In fact, it is known that relaxing this requirement results in algorithms with smaller space-complexity.

In this paper, we are aiming at measuring how much gain in terms of memory can be expected by using arbitrary deterministic self-stabilizing algorithms, not necessarily silent. To our knowledge, the only known lower bound on the memory requirement for deterministic general algorithms, also established at the end of the 90's, is due to Beauquier et al. [PODC '99] who proved that registers of constant size are not sufficient for leader election algorithms. We improve this result by establishing the lower bound $\Omega(\log \log n)$ bits per register for deterministic self-stabilizing algorithms solving $(\Delta + 1)$-coloring, leader election or constructing a spanning tree in networks of maximum degree Δ.

2012 ACM Subject Classification Theory of computation → Distributed computing models

Keywords and phrases Space lower bound, memory tight bound, deterministic self-stabilization, leader election, anonymous, identifiers, state model, ring

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.37

Related Version `http://arxiv.org/abs/1905.08563`

Funding Support by ANR ESTATE.

1 Introduction

Self-stabilization is a suitable paradigm for asynchronous distributed systems subject to transient failures. The occurrence of failures can bring the system into arbitrary configurations. A self-stabilizing algorithm guarantees a return to a correct behavior in finite time, without external intervention. The legality of the configuration is a notion that depends on the problem considered. For instance, for the problem of $(\Delta + 1)$-coloration of the nodes, a configuration is legal if every node has a color in $[1, \ldots, \Delta + 1]$ different from the color of each of its neighbors.

© Lélia Blin, Laurent Feuilloley, and Gabriel Le Bouder;
licensed under Creative Commons License CC-BY
33rd International Symposium on Distributed Computing (DISC 2019).
Editor: Jukka Suomela; Article No. 37; pp. 37:1–37:3

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In the state model, the processes have two types of memory. The *immutable* memory is used to store the identity of the node, its ports numbers, and the code of the algorithm. By definition the immutable memory is fault free. On the other hand, the *mutable* memory is the memory used to store variables, this memory is not fault free. In fact, each node has only one register and the register of each node is composed by the mutable memory. Recall that in this model, each node uses its register to communicate the values of its own variables to its neighbors. As a result, only mutable memory (a.k.a register) is considered for computing *memory complexity* a.k.a *space complexity* of a self-stabilizing protocol, and immutable memory is not. Note that a node can use to compute the information of its immutable memory, such as its identity.

Indeed, small space-complexity is desirable for several reasons. One reason is for reducing the overhead due to *link congestion* [1] (note that the nodes carry on exchanging information, even after stabilization, and even when no fault occurs). Another reason is that mixing *variable replication* with self-stabilization can be desirable [8], but replication is possible only if the overall memory occupied by these variables is small.

In this paper, we establish a lower bound of $\Omega(\log\log n)$ bits space memory for the register of each node for the leader election problem, the $(\Delta+1)$-coloration of the nodes (denoted by *coloring* problem), and the spanning tree construction. Note that we consider a network where each node $v \in V$ has a distinct identity, denoted by $ID(v) \in \{1, \dots, n^c\}$ for some constant $c > 1$. This significantly improves the only lower bound [2] known so far, from $\Omega(1)$ to $\Omega(\log\log n)$. Moreover, our lower bound invalidates the folklore conjecture stating that the aforementioned problems might be solvable using only $O(\log^* n)$ bits by register. More importantly, our lower bound implies that the upper bound $O(\log\Delta + \log\log n)$ bits of register per node in [6] for the *coloring* problem and spanning tree construction is space optimal.

In the context of self-stabilization, a *silent* algorithm guarantees that the register of every node does not change once the algorithm has stabilized. The memory efficiency in the context of silent self-stabilizing algorithms is well studied. Firstly, Dolev and al. [7], at the end of the 90's, proved that finding the centers of a graph, leader election, and spanning tree construction require a memory of $\Omega(\log n)$ bits per register whenever the algorithm is requested to be silent. On the other hand, the design of silent algorithms is based on a mechanism known as *proof-labeling-scheme* (PLS) [10], and space lower bounds for PLSs imply space lower bounds for silent self-stabilizing algorithms. A typical example is the $\Omega(\log^2 n)$-bit lower bound on the size of every PLS for minimum-weight spanning trees (MST) [9] , which implies the same bound for constructing an MST in a silent manner [4]. It follows that requiring the algorithm to be silent has a cost in terms of memory space, while, in the context of self-stabilization, where every node constantly checks the states of its neighbors, the silence property can be of no practical interest. In fact, it is known that relaxing this requirement results in algorithms with smaller space-complexity [5, 6].

2 Lower bounds

2.1 Main theorem

▶ **Theorem 1.** *Let $c > 1$. Every self-stabilizing deterministic algorithm under the fair distributed scheduler solving $(\Delta+1)$-coloring, leader election, or spanning tree construction in n-node graphs where every node has a unique identities in $[1, n^c]$ requires $\Omega(\log\log n)$ bits of memory per register at each node.*

The proof is to be found in the full version [3]. The core of the proof is the following. It is known that problems such as coloring, leader election and spanning tree construction require identities to break symmetry under a distributed scheduler. In other words, no algorithm

can solve such problems in anonymous networks. In essence, we show that an algorithm in a network with identities but in which the size of each register is too small does not have more power than an algorithm running in an anonymous network. More specifically, let A be an algorithm in a network with identities, and let us assume that A uses $o(\log \log n)$ bits by register. Observe that even if the network has identities, and even if these identities are used by the algorithm, their size is not taken into account in the space complexity of the algorithm, as identities are not stored in register, but in the immutable memory. However, the nodes cannot write their identities in the register, which is too small, and the identities have to be transferred between nodes in a series of smaller pieces of information. We show that, with only $o(\log \log n)$ bits node register, there exist graphs and identity assignments to the nodes of these graphs such that the algorithm A has the same behavior as an algorithm in the anonymous version of these graphs. It follows that A cannot solve coloring problem, leader election and spanning tree construction.

Observe that it is usually assumed that any coloring algorithm must compute a color variable $c_v \in [1, \dots, \Delta + 1]$ at each node v. It follows that such coloring algorithms require registers of size $\Omega(\log \Delta)$ bits. Similarly, it is usually assumed that any algorithm constructing a spanning tree computes a port number $p_v \in [1, \dots, \Delta]$ at each node $v \in V$. Hence, such algorithms require registers of size $\Omega(\log \Delta)$ bits. Consequently, the deterministic self-stabilizing algorithms for coloring and for spanning tree construction in [6] are space-optimal. Indeed, these algorithms use $O(\log \Delta + \log \log n)$ bits of memory at each node.

References

1. J. Adamek, M. Nesterenko, and S. Tixeuil. Evaluating Practical Tolerance Properties of Stabilizing Programs through Simulation: The Case of Propagation of Information with Feedback. In *SSS 2012*, pages 126–132, 2012.
2. J. Beauquier, M. Gradinariu, and C. Johnen. Memory Space Requirements for Self-Stabilizing Leader Election Protocols. In *PODC 1999*, pages 199–207, 1999.
3. L. Blin, L. Feuilloley, and G. Le Bouder. Memory Lower Bounds for Self-Stabilization. *CoRR*, abs/1905.08563, 2019. `arXiv:1905.08563`.
4. L. Blin and P. Fraigniaud. Space-Optimal Time-Efficient Silent Self-Stabilizing Constructions of Constrained Spanning Trees. In *ICDCS 2015*, pages 589–598, 2015.
5. L. Blin and S. Tixeuil. Compact deterministic self-stabilizing leader election on a ring: the exponential advantage of being talkative. *Distributed Computing*, 31(2):139–166, 2018.
6. L. Blin and S. Tixeuil. Compact Self-Stabilizing Leader Election for General Networks. In *LATIN 2018*, pages 161–173, 2018.
7. S. Dolev, M. G. Gouda, and M. Schneider. Memory Requirements for Silent Stabilization. *Acta Inf.*, 36(6):447–462, 1999.
8. T. Herman and S. V. Pemmaraju. Error-detecting codes and fault-containing self-stabilization. *Inf. Process. Lett.*, 73(1-2):41–46, 2000.
9. A. Korman and S. Kutten. Distributed verification of minimum spanning trees. *Distributed Computing*, 20(4):253–266, 2007.
10. A. Korman, S. Kutten, and D. Peleg. Proof labeling schemes. *Distributed Computing*, 22(4):215–233, 2010.

Brief Announcement: Wait-Free Universality of Consensus in the Infinite Arrival Model

Grégoire Bonin
LS2N, Université de Nantes, France
gregoire.bonin@etu.univ-nantes.fr

Achour Mostéfaoui
LS2N, Université de Nantes, France
achour.mostefaoui@univ-nantes.fr

Matthieu Perrin
LS2N, Université de Nantes, France
matthieu.perrin@univ-nantes.fr

Abstract

In classical asynchronous distributed systems composed of a fixed number n of processes where some proportion may fail by crashing, many objects do not have a wait-free linearizable implementation (e.g. stacks, queues, etc.). It has been proved that consensus is universal in such systems, which means that this system augmented with consensus objects allows to implement any object that has a sequential specification. In this paper, we consider a more general system model called infinite arrival model where infinitely many processes may arrive and leave or crash during a run. We prove that consensus is still universal in this more general model. For that, we propose a universal construction based on a weak log that can be implementated using consensus objects.

2012 ACM Subject Classification Theory of computation → Distributed computing models

Keywords and phrases Concurrent object, Consensus, Infinite arrival model, Linearizability, Universal construction, Wait-freedom

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.38

Related Version A full version of the paper is available at `https://arxiv.org/abs/1908.02063`.

Funding This work was partially supported by the French ANR project 16-CE25-0005 O'Browser devoted to the study of decentralized applications on Web browsers.

1 Introduction

Maurice Herlihy proved in [3] that consensus is universal in classical distributed systems composed of a set of n processes. Namely, any object having a sequential specification has a wait-free and linearizable implementation using only read/write registers (memory locations) and some number of consensus objects. For proving the universality of consensus, Herlihy introduced the notion of universal construction. It is a generic algorithm that, given a deterministic sequential specification of any object, provides a concurrent implementation of this object. Since then, many universal constructions have been proposed for several objects [5], assuming the availability of hardware special instructions that provide the same computing power as consensus, like compare&swap, Load-Link/Store-Conditional etc.

This last decade, first with peer-to-peer systems, and then with multi-core machines and the multi-threading model, the assumption of a closed system with a fixed number n of processes and where every process knows the identifiers of all processes became too restrictive. Hence the infinite arrival model introduced in [4]. In this model, any number of processes can crash (or leave, in a same way as in the other model), but any number (be it finite or not) of processes can also join the network. When a process joins such a system, it is not

© Grégoire Bonin, Achour Mostéfaoui, and Matthieu Perrin;
licensed under Creative Commons License CC-BY

known to the already running processes, so no fixed number of processes can be used in the implementations as a parameter. Let us note that, at any time, the number of processes that have already joined the system is finite, but can be infinitely growing.

Problem statement. The aim of this paper is to extend universality of consensus to the infinite arrival model. The question is thus "is it possible to build a universal wait-free linearizable construction based on consensus objects and read/write atomic registers?" This is not trivial for different reasons. First, although the lock-free universal constructions still work in the infinite arrival model because they ensure a global progress condition, this is no more the case for wait-free universal constructions. Second, wait-free implementations rely on what is called help mechanism, that has been recently formalized in [1]. This mechanism requires any process, before terminating its operation, to help processes having pending operations, in order to reach wait-freedom. One of the difficulties in the infinite arrival model is that helping is not obvious. Indeed, helping requires at least that a process needing to be helped is able to announce its existence to other processes willing to help it. Due to the infinite number of potential participating processes over time, it is not reasonable to assume that each process can write in a dedicated register, and to require helping processes to read them all. When only consensus and read/write registers are accessible to a process, a newly arriving process must compete with a potentially infinite number of other arriving processes on either a consensus object or a same memory location; and may fail on all its attempts.

2 The Weak Log Abstraction

Similarly to [2] which first proposes a Collect object that will be used as a building block for a universal construction, we propose a weak log object that is used as a list of presence where a process that arrives registers. A weak log can then be used in Herlihy's universal construction [3] instead of the array of registers to achieve wait-freedom. In an instance of the weak log, each process p_i proposes a value through an operation $\texttt{append}(v_i)$, that returns the sequence of all the values previously appended. The weak log is wait-free but not linearizable. Instead, it is specified by the following properties.

▶ **Definition 1** (weak log). *All processes $p_0, p_1, \dots$ propose distinct values $v_0, v_1, \dots$ by invoking $\texttt{append}(v_i)$, that returns a finite sequence $w_i = w_{i,1} \cdot w_{i,2} \cdots w_{i,|w_i|}$ such that:*

Validity. *All values in a sequence w_i have been appended by some process.*

Suffixing. *The last value of the sequence returned by p_i is its own.*

Total order. *All pairs of values contained in both w_i and w_j appear in the same order.*

Eventual visibility. *If p_i terminates, finitely many returned sequences do not contain v_i.*

Wait-freedom. *No process takes an infinite number of steps in an execution.*

The main difficulty in the implementation of a weak log lies in the allocation of one memory location per process, where it can safely announce its invoked operation. As it is impossible to allocate an infinite array at once, it is necessary to build a data structure in which processes allocate their own piece of memory, and make it reachable to other processes, by winning a consensus. A linked list in which processes compete to append their value at the end follows a similar pattern, but it poses a challenge: as an infinite number of processes access the same sequence of consensus objects, one process may loose all its attempts to insert its own node, breaking wait-freedom.

Algorithm 1 solves this issue by using a novel feature, that we call *passive helping*: when a process wins a consensus, it creates a side list to host values of processes concurrently competing on the same consensus object. As only a finite number of processes have arrived

Algorithm 1 Wait-free weak log using consensus.

```
1 operation append(v) is:
2   node_i ← last.read().propose(⟨⟨v, ⊥⟩, ⊥⟩) ;                    // add v to the log
3   last.write(node_i.tail);
4   while node_i.head ≠ v do  node_i ← node_i.tail.propose(⟨v, ⊥⟩);
5   log_i ← ε; list_i ← first; node_i ← list_i.head;                  // read the log
6   while true do
7     log_i ← log_i ⊕ node_i.head;
8     if node_i.head = v then return log_i;
9     node_i ← node_i.tail; if node_i = ⊥ then  list_i ← list_i.tail; node_i ← list_i.head;
```

in the system when the consensus is won, a finite number of processes will try to insert their value in the side list, which ensures termination.

Processes executing Algorithm 1 build a linked list of linked lists of nodes of the form ⟨list.head, list.tail⟩ where list.tail is a consensus object that references nodes of the same form, and list.head = ⟨node.head, node.tail⟩ is a node of the side list, where node.head is a value appended by some process and node.tail is a consensus object accepting values of the same type as list.head. Processes share a consensus object, first, that references the first node of the list of lists, and a read/write register, last, that references a consensus object list.tail.

In absence of concurrency, last references the end of the list starting with first. However, as the consensus and the write on lines 2 and 3 are not done atomically, a very old value can be written in last, in which case its value could move backward. The central property of the algorithm is that last eventually moves forward, allowing very slow processes to find some place in a side list.

3 Conclusion

Consensus is a central problem in distributed computing, because it allows wait-free linearizable implementations of all objects with a sequential specification, in systems composed of n asynchronous processes that may crash. In this paper, we asked the question of whether the result still holds in the infinite arrival model, in which a potentially infinite number of processes can arrive and leave during an execution. We answered this question positively by introducing a weak log abstraction, that can be implemented using only consensus objects and read/write registers and can be used in a wait-free and linearizable universal construction.

References

1. Keren Censor-Hillel, Erez Petrank, and Shahar Timnat. Help! In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 241–250. ACM, 2015.
2. Panagiota Fatourou and Nikolaos D. Kallimanis. Highly-Efficient Wait-Free Synchronization. *Theory Comput. Syst.*, 55(3):475–520, 2014.
3. Maurice Herlihy. Impossibility and Universality Results for Wait-Free Synchronization. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing, Toronto, Ontario, Canada, August 15-17, 1988*, pages 276–290, 1988.
4. Michael Merritt and Gadi Taubenfeld. Resilient consensus for infinitely many processes. In *International Symposium on Distributed Computing*, pages 1–15. Springer, 2003.
5. Michel Raynal. Distributed Universal Constructions: a Guided Tour. *Bulletin of the EATCS*, 121, 2017.

Brief Announcement: Asymmetric Distributed Trust

Christian Cachin

University of Bern, Switzerland
https://crypto.unibe.ch/cc/
cachin@inf.unibe.ch

Björn Tackmann

DFINITY, Zurich, Switzerland
bjoern.tackmann@alumni.ethz.ch

Abstract

Quorum systems are a key abstraction in distributed fault-tolerant computing for capturing trust assumptions. They can be found at the core of many algorithms for implementing reliable broadcasts, shared memory, consensus and other problems. This paper introduces *asymmetric Byzantine quorum systems* that model subjective trust. Every process is free to choose which combinations of other processes it trusts and which ones it considers faulty. Asymmetric quorum systems strictly generalize standard Byzantine quorum systems, which have only one global trust assumption for all processes. This work also presents protocols that implement abstractions of shared memory and broadcast primitives with processes prone to Byzantine faults and asymmetric trust. The model and protocols pave the way for realizing more elaborate algorithms with asymmetric trust.

2012 ACM Subject Classification Theory of computation → Cryptographic protocols; Software and its engineering → Distributed systems organizing principles

Keywords and phrases Quorums, consensus, distributed trust, blockchains, cryptocurrencies

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.39

Related Version A full version of the paper is available athttp://arxiv.org/abs/1906.09314.

Acknowledgements This work has been supported in part by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 780477 PRIViLEDGE.

1 Extended Abstract

Byzantine quorum systems [4] are a fundamental primitive for building resilient distributed systems from untrusted components. Given a set of nodes, a quorum system captures a trust assumption on the nodes in terms of potentially malicious protocol participants and colluding groups of nodes. Quorum systems are at the core of many distributed programming abstractions.

Traditionally, trust in a Byzantine quorum system for a set of processes $\mathcal{P}$ has been *symmetric*. In other words, a global assumption specifies which processes may fail, such as the simple and prominent *threshold quorum* assumption, in which any subset of $\mathcal{P}$ of a given maximum size may collude and act against the protocol. The most basic threshold Byzantine quorum system, for example, allows all subsets of up to $f < n/3$ processes to fail. Some classic works also model arbitrary, non-threshold symmetric quorum systems [4, 3].

However, trust is inherently subjective. *De gustibus non est disputandum.* Estimating which processes will function correctly and which ones will misbehave may depend on personal taste. A myriad of local choices influences one process' trust in others, especially because there are so many forms of "malicious" behavior. Some processes might not even be aware of all others, yet a process should not depend on unknown third parties in a distributed

© Christian Cachin and Björn Tackmann;
licensed under Creative Commons License CC-BY

33rd International Symposium on Distributed Computing (DISC 2019).
Editor: Jukka Suomela; Article No. 39; pp. 39:1–39:3

collaboration. How can one model asymmetric trust in distributed protocols? Can traditional Byzantine quorum systems be extended to subjective failure assumptions? How do the standard protocols generalize to this model?

In this paper, we answer these questions and introduce models and protocols for asymmetric distributed trust. We formalize *asymmetric quorum systems* for asynchronous protocols, in which every process can make its own assumptions about Byzantine faults of others. We introduce several protocols with asymmetric trust that strictly generalize the existing algorithms, which require common trust.

Interest in consensus protocols based on Byzantine quorum systems has surged recently because of their application to permissioned blockchain networks [2]. A middle ground between permissionless blockchains based on Proof-of-Work protocols and permissioned ones has been introduced by the blockchain networks of Ripple (`https://ripple.com`) and Stellar (`https://stellar.org`). Their stated model for achieving network-level consensus uses subjective trust in the sense that each process declares a local list of processes that it "trusts" in the protocol.

Consensus in the *Ripple* blockchain is executed by its validator nodes. Each validator declares a *Unique Node List (UNL)*, which is a "list of transaction validators a given participant believes will not conspire to defraud them;" but on the other hand, "Ripple provides a default and recommended list which we expand based on watching the history of validators operated by Ripple and third parties." Many questions have therefore been raised about the kind of decentralization offered by the Ripple protocol. This debate has not yet been resolved. *Stellar* was created as an evolution of Ripple that shares much of the same design philosophy. The Stellar consensus protocol [5] introduces *federated Byzantine quorum systems (FBQS)*; these bear superficial resemblance with our asymmetric quorum systems but differ technically. Stellar's consensus protocol uses *quorum slices*, which are "the subset of a quorum that can convince one particular node of agreement." In an FBQS, "each node chooses its own quorum slices" and "the system-wide quorums result from these decisions by individual nodes". However, standard Byzantine quorum systems and FBQS are *not* comparable because (1) an FBQS when instantiated with the same trust assumption for all processes does not reduce to a symmetric quorum system and (2) existing protocols do not generalize to FBQS.

Understanding how such ideas of subjective trust, as manifested in the Ripple and Stellar blockchains, relate to traditional quorum systems is the main motivation for this work. Our protocols for asymmetric trust generalize the well-known, classic algorithms in the literature and therefore look superficially similar. They are much more powerful, however.

The contributions as detailed in the full paper [1] are as follows:

- We introduce asymmetric Byzantine quorum systems formally as an extension of standard Byzantine quorum systems and discuss some of their properties.
- We show two implementations of a shared register, with single-writer, multi-reader regular semantics, using asymmetric Byzantine quorum systems.
- We examine broadcast primitives in the Byzantine model with asymmetric trust. In particular, we define and implement Byzantine consistent and reliable broadcast protocols. The latter primitive is related to a "federated voting" protocol used by Stellar consensus [5].

Asymmetric quorum systems

Consider a system of n *processes* $\mathcal{P} = \{p_1, \ldots, p_n\}$ that communicate with each other. *Byzantine quorum systems* have been introduced by Malkhi and Reiter [4] with respect to a *fail-prone system* $\mathcal{F} \subseteq 2^{\mathcal{P}}$, a collection of subsets of $\mathcal{P}$, none of which is contained in another,

such that some $F \in \mathcal{F}$ with $F \subseteq \mathcal{P}$ is called a *fail-prone set* and contains all processes that may at most fail together in some execution [4]. A fail-prone system captures an assumption on the possible failure patterns that may occur. We let $\mathcal{A}^* = \{A' | A' \subseteq A, A \in \mathcal{A}\}$.

▶ **Definition 1.** *A* Byzantine quorum system *for $\mathcal{F}$ is a collection of sets of processes $\mathcal{Q} \subseteq 2^{\mathcal{P}}$, where each $Q \in \mathcal{Q}$ is called a* quorum*, such that the following properties hold:*

Consistency: *The intersection of any two quorums contains at least one process that is not faulty.*

Availability: *For any set of processes that may fail together, there exists a disjoint quorum in $\mathcal{Q}$.*

This is also known as a *Byzantine dissemination quorum system* [4]. The Q^3-condition [4, 3] generalizes the assumption that $n > 3f$ are needed to tolerate f faulty ones in Byzantine protocols. A fail-prone system $\mathcal{F}$ satisfies the Q^3*-condition*, abbreviated as $Q^3(\mathcal{F})$, whenever it holds $\forall F_1, F_2, F_3 \in \mathcal{F} : \mathcal{P} \not\subseteq F_1 \cup F_2 \cup F_3$. It is well-known [4] that a Byzantine quorum system for $\mathcal{F}$ exists if and only if $Q^3(\mathcal{F})$.

With asymmetric trust, every process is free to make its own trust assumption and to express this with a fail-prone system. Hence, an *asymmetric fail-prone system* $\mathbb{F} = [\mathcal{F}_1, \ldots, \mathcal{F}_n]$ consists of an array of fail-prone systems, where $\mathcal{F}_i$ denotes the trust assumption of p_i.

▶ **Definition 2.** *An* asymmetric Byzantine quorum system *for $\mathbb{F}$ is an array of collections of sets $\mathbb{Q} = [\mathcal{Q}_1, \ldots, \mathcal{Q}_n]$, where $\mathcal{Q}_i \subseteq 2^{\mathcal{P}}$ for $i \in [1, n]$. The set $\mathcal{Q}_i \subseteq 2^{\mathcal{P}}$ is called the* quorum system of p_i *and any set $Q_i \in \mathcal{Q}_i$ is called a* quorum (set) for p_i. *It satisfies:*

Consistency: *The intersection of two quorums for any two processes contains at least one process for which both processes assume that it is not faulty, i.e., $\forall i, j \in [1, n], \forall Q_i \in \mathcal{Q}_i, \forall Q_j \in \mathcal{Q}_j, \forall F_{ij} \in {\mathcal{F}_i}^* \cap {\mathcal{F}_j}^* : Q_i \cap Q_j \not\subseteq F_{ij}$.*

Availability: *For any process p_i and any set of processes that may fail together according to p_i, there exists a disjoint quorum for p_i in $\mathcal{Q}_i$, i.e., $\forall i \in [1, n], \forall F_i \in \mathcal{F}_i : \exists Q_i \in \mathcal{Q}_i : F_i \cap Q_i = \emptyset$.*

The existence of asymmetric quorum systems can be characterized with a property that generalizes the Q^3-condition for the underlying asymmetric fail-prone systems as follows. Namely, an asymmetric fail-prone system $\mathbb{F}$ satisfies the B^3*-condition*, abbreviated as $B^3(\mathbb{F})$, whenever it holds that $\forall i, j \in [1, n], \forall F_i \in \mathcal{F}_i, \forall F_j \in \mathcal{F}_j, \forall F_{ij} \in {\mathcal{F}_i}^* \cap {\mathcal{F}_j}^* : \mathcal{P} \not\subseteq F_i \cup F_j \cup F_{ij}$.

▶ **Theorem 3.** *An asymmetric fail-prone system $\mathbb{F}$ satisfies $B^3(\mathbb{F})$ if and only if there exists an asymmetric quorum system for $\mathbb{F}$.*

References

1 Christian Cachin and Björn Tackmann. Asymmetric Distributed Trust. e-print, arXiv:1906.09314 [cs.DC], 2019. arXiv:1906.09314.

2 Christian Cachin and Marko Vukolic. Blockchain Consensus Protocols in the Wild. In Andréa W. Richa, editor, *Proc. 31st Intl. Symposium on Distributed Computing (DISC 2017)*, volume 91 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:16, 2017. doi:10.4230/LIPIcs.DISC.2017.1.

3 Martin Hirt and Ueli Maurer. Player Simulation and General Adversary Structures in Perfect Multi-Party Computation. *Journal of Cryptology*, 13(1):31–60, 2000.

4 Dahlia Malkhi and Michael K. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

5 David Mazières. The Stellar Consensus Protocol: A Federated Model for Internet-level Consensus. Stellar, available online, https://www.stellar.org/papers/stellar-consensus-protocol.pdf, 2016.

Brief Announcement: Implementing Byzantine Tolerant Distributed Ledger Objects

Vicent Cholvi
Universitat Jaume I, Castellón de la Plana, Spain
vcholvi@uji.es

Antonio Fernández Anta
IMDEA Networks Institute, Madrid, Spain
antonio.fernandez@imdea.org

Chryssis Georgiou
University of Cyprus, Nicosia, Cyprus
chryssis@cs.ucy.ac.cy

Nicolas Nicolaou
Algolysis Ltd, Lemesos, Cyprus
nicolas@algolysis.com

Abstract

This work provides a proper formalization for Distributed Ledger Objects (as first defined in [1]), when processes may be Byzantine. The formal definitions are accompanied by algorithms to implement Byzantine Distributed Ledgers by utilizing a Byzantine Atomic Broadcast service.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Distributed Ledger Object, Byzantine Faults

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.40

Funding This work was co-funded by the European Regional Development Fund and the Republic of Cyprus through the Research Promotion Foundation (Project: POST-DOC/0916/0090), by research funds from the University of Cyprus (CG-RA2019), by the Spanish grant TIN2017-88749-R (DiscoEdge), the Region of Madrid EdgeData-CM program (P2018/TCS-4499), the NSF of China grant 61520106005, and by the Spanish Ministerio de Educación Cultura y Deporte under grant PRX18/00163.

1 Introduction

The work in [1] introduced the notion of a *Distributed Ledger Object* (DLO) in an attempt to provide a formalization of Distributed Ledgers (blockchains) from a Distributed Computing point of view. A DLO is a concurrent shared object that stores a totally ordered sequence of *records*, and supports two operations: *append* and *get*. A record can be seen as an abstraction of a transaction or a block of transactions. As operations may access the DLO concurrently, the work in [1] defines eventual, sequential, and linearizable consistency gurantees for DLOs. These formalisms were independent of the communication medium (message-passing or shared-memory) and the timing model (synchrony or asynchrony). Three DLO implementations, one for each consistency guarantee, were specified in [1] for a message-passing asynchronous model, assuming that clients and servers may crash. However, in existing blockchain systems, both the servers (e.g., miners) and the clients (e.g., users) could be acting maliciously. To this respect, in this work we propose implementations where *both* the clients and the servers can be Byzantine, i.e., we propose implementations of *Byzantine Tolerant* linearizable DLOs.

© Vicent Cholvi, Antonio Fernández Anta, Chryssis Georgiou, and Nicolas Nicolaou;
licensed under Creative Commons License CC-BY
33rd International Symposium on Distributed Computing (DISC 2019).
Editor: Jukka Suomela; Article No. 40; pp. 40:1–40:3

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2 Model

Distributed Ledger Objects. A Distributed Ledger Object (DLO) is a concurrent object that stores a totally ordered sequence of records (initially empty). A DLO $\mathcal{L}$ supports two operations, $\mathcal{L}.append()$ and $\mathcal{L}.get()$, which append a new record to the sequence and return the whole sequence, respectively [1]. The DLO is implemented by a set of servers that collaborate running a distributed algorithm. The DLO is used by a set of clients that access it by invoking append and get operations, which are translated into request and response messages exchanged with the servers. An operation π is *complete* in an execution ξ, if both the request and matching response of π appear in ξ. We say that an operation π_1 *precedes* an operation π_2, or π_2 *succeeds* π_1, in an execution ξ if the response event of π_1 appears before the invocation event of π_2 in ξ; otherwise the two operations are *concurrent*.

Failure Model. In this work we assume that processes (servers and clients) can fail arbitrarily, i.e., we assume that failures are Byzantine. Hence, we assume a *Byzantine system* in which *any number of clients,* and *up to f servers* can fail arbitrarily. The total number of servers is at least $3f + 1$. We also assume that the messages sent by any process (server or client) are authenticated, so that messages corrupted or fabricated by Byzantine processes are detected and discarded by correct processes [3]. Communication channels between correct processes are reliable but asynchronous.

Byzantine-tolerant DLO. This paper aims to propose algorithms that implement a linearizable DLO $\mathcal{L}$ in Byzantine systems. Since Byzantine clients and server can behave arbitrarily, we define the properties that a DLO must satisfy adapted to Byzantine systems. In particular, since Byzantine processes may return any arbitrary sequence or append any record, the properties only consider the actions of *correct processes*.

- *Byzantine Strong Prefix (BSP):* If two *correct clients* issue two $\mathcal{L}.get()$ operations that return record sequences S and S' respectively, then either S is a prefix of S' or vice-versa.
- *Byzantine Linearizability (BL):* Let G be the set of all complete get operations issued by correct clients. Let A be the set of complete append operations $\mathcal{L}.append(r)$ such that $r \in S$ and S is the sequence returned by some operation $\mathcal{L}.get() \in G$. Then linerizability holds with respect to the set of operations $G \cup A$. This property is similar to the one described in [5] for registers.

In the remainder we say that a DLO is *Byzantine Tolerant* if it satisfies the properties BSP and BL in a Byzantine system. Observe that DLOs are oblivious to the syntax and semantics of the records they hold [1]. Hence, in this paper we do not need to care about whether the records appended by a Byzantine client are syntactically and semantically valid.

Byzantine Atomic Broadcast: In the algorithms we propose in this paper we use a Byzantine Atomic Broadcast (BAB) service for the server communication [2, 3, 4], that satisfies the following properties: validity, agreement, integrity and total order. Note that the work in [1] utilized a crash-tolerant Atomic Broadcast (AB) service to implement a crash-tolerant DLO. The properties assumed here for the BAB service are similar to their counterpart in the AB service, but applied only to correct processes. Despite the use of a BAB in this work, additional machinery is required in order to implement a Byzantine DLO and ensure the satisfaction of properties BSP and BL.

3 Algorithms for Byzantine-tolerant DLOs

Algorithm 1 API to the operations of a DLO $\mathcal{L}$, executed by Client p.

```
1:  Init: c ← 0
2:  function L.get( )
3:      c ← c + 1
4:      send request (c, p, GET) to ≥ 2f + 1 servers
5:      wait resp. (c, i, GETRESP, S) from f + 1
    different servers with the same sequence S
6:      return S
7:  function L.append(r)
8:      c ← c + 1
9:      send request (c, p, APPEND, r) to at least
    2f + 1 different servers
10:     wait resp. (c, i, APPENDRESP, ACK) from f+1
    different servers
11:     return ACK
```

Algorithm 2 Byzantine-tolerant DLO; Code for Server i.

```
1:  Init: S_i ← ∅
2:  receive (c, p, GET) from process p
3:      BAB-broadcast(c, p, GET, i)
4:  upon (BAB-deliver(c, p, GET, j)) do
5:      if ((c, p, GET, -) has been BAB-delivered f+1
    times from different servers) then
6:          send resp. (c, i, GETRESP, S_i) to p
7:  receive (c, p, APPEND, r) from process p
8:      BAB-broadcast(c, p, APPEND, r, i)
9:  upon (BAB-deliver(c, p, APPEND, r, j)) do
10:     if (r ∉ S_i) and ((c, p, APPEND, r, -) has been
    BAB-delivered at f + 1 different servers) then
11:         S_i ← S_i‖r
12:         send resp. (c, i, APPENDRESP, ACK) to p
```

Client Algorithm. The algorithm executed by a client that invokes a get or append operation on a DLO $\mathcal{L}$ is presented in Code 1. An operation starts when the corresponding function of Code 1 is invoked, and it ends when the matching **return** instruction is executed. A Byzantine client p may not follow Code 1 (as it may behave arbitrarily) but still be able to append a record r in the ledger. So, some correct client may obtain, in the response to a get operation, a sequence that contains r.

When an operation is invoked, a correct client increments a local counter and then sends operation requests to a set of at least $2f+1$ servers, to guarantee that at least $f+1$ correct servers receive it. A get operation is completed when the client receives $f+1$ *consistent* replies and an append is completed when the client receives $f+1$ replies from different servers. Both cases guarantee the response from at least one correct server.

Server Algorithm. The algorithm executed by the servers is presented in Code 2. The algorithm uses the Byzantine Atomic Broadcast service to impose a total order in the messages shared among the servers. Operations received from clients are BAB-broadcast using this service, which are eventually BAB-delivered. An operation is processed by a server only when it has been BAB-delivered $f+1$ times (sent by different servers). This implies that at least one correct server sent it. The properties of the BAB service guarantee that all correct servers receive the same sequence of messages BAB-delivered, and hence process the operations at the same point, maintaining their states consistent.

▶ **Theorem 1.** *The combination of the algorithms presented in Codes 1 and 2 implements a linearizable Byzantine Tolerant distributed ledger object.*

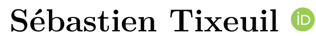
References

1. Antonio Fernández Anta, Kishori M. Konwar, Chryssis Georgiou, and Nicolas C. Nicolaou. Formalizing and Implementing Distributed Ledger Objects. *SIGACT News*, 49(2):58–76, 2018.
2. Paulo Coelho, Tarcisio Ceolin Junior, Alysson Bessani, Fernando Dotti, and Fernando Pedone. Byzantine Fault-Tolerant Atomic Multicast. In *DSN 2018*, pages 39–50. IEEE, 2018.
3. F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement. *Information and Computation*, 118(1):158–179, 1995.
4. Zarko Milosevic, Martin Hutle, and André Schiper. On the Reduction of Atomic Broadcast to Consensus with Byzantine Faults. In *SRDS 2011*, pages 235–244, 2011.
5. Achour Mostéfaoui, Matoula Petrolia, Michel Raynal, and Claude Jard. Atomic Read/Write Memory in Signature-Free Byzantine Asynchronous Message-Passing Systems. *Th. Comp. Syst.*, 60(4):677–694, 2017.

Brief Announcement: Model Checking Rendezvous Algorithms for Robots with Lights in Euclidean Space

Xavier Défago
School of Computing, Tokyo Institute of Technology, Japan

Adam Heriban
Sorbonne Université, CNRS, LIP6, Paris, France


Sébastien Tixeuil
Sorbonne Université, CNRS, LIP6, Paris, France

Koichi Wada
Faculty of Science and Engineering, Hosei University, Tokyo, Japan

Abstract

This announces the first successful attempt at using model-checking techniques to verify the correctness of self-stabilizing distributed algorithms for robots evolving in a *continuous* environment. The study focuses on the problem of rendezvous of two robots with lights and presents a generic verification model for the SPIN model checker. It will be presented in full at an upcoming venue.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Theory of computation → Verification by model checking; Computer systems organization → Robotic autonomy; Theory of computation → Self-organization

Keywords and phrases Autonomous mobile robots, Rendezvous, Lights, Model Checking

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.41

Related Version Details of the study hosted on arXiv [2] at `https://arxiv.org/abs/1907.09871`.

Funding This work was supported by JST SICORP Grant Number JPMJSC1606, JST SICORP Grant Number JPMJSC1806, and JSPS KAKENHI Grant Number 17K00019.

Introduction

Following the seminal work of Suzuki and Yamashita [9], we model robots as points in the 2D Euclidean plane that independently execute their own instance of the same deterministic algorithm. Robots are anonymous, oblivious, and disoriented (i.e., no common coordinate system), and repeatedly execute Look-Compute-Move (LCM) cycles, where a robot "Looks" at its surroundings, obtains a snapshot of the locations of all robots, "Computes" a destination, and "Moves" toward it. Additionally, robots are equipped with a light that can emit a color among a fixed number of distinct colors [1]. A robot observes all lights during its Look phase and changes its light at the end of its Compute phase.

The literature [9, 3] considers three main levels of synchrony: FSYNC model [9] where every LCM cycle is performed simultaneously by all robots; SSYNC where each cycle may be skipped by a subset of the robots; and ASYNC which imposes no restriction on synchronization between the robots, thus allowing robots to be observed while moving.

The problem of gathering requires all robots to reach a single point in finite time, regardless of their initial locations. The particular case of gathering two robots is called *rendezvous*. In SSYNC, no deterministic algorithm can solve rendezvous without additional assumptions [9]. This case being trickier than three or more robots due to the inherent symmetry of observed configurations. A rendezvous algorithm is self-stabilizing if robots eventually reach and

© Xavier Défago, Adam Heriban, Sébastien Tixeuil, and Koichi Wada;
licensed under Creative Commons License CC-BY
33rd International Symposium on Distributed Computing (DISC 2019).
Editor: Jukka Suomela; Article No. 41; pp. 41:1–41:3

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

stay forever at the same location regardless of the *initial configuration*. Algorithms that set constraints on the initial configuration (*e.g.*, must start with a specific color) are not self-stabilizing.

Viglietta [10] proved that two colors are sufficient in SSYNC and that three colors are necessary and sufficient in ASYNC for a specific class of algorithms. Okumuraet al. [8] presented rendezvous algorithms when additional restrictions are made on the model (rigid moves, initial colors, etc.). Finally, Heribanet al. [5] showed that two colors are necessary and sufficient in ASYNC without extra assumptions.

As they use case-based reasoning, the handwritten proofs of rendezvous algorithms with lights are lengthy, complex, tedious to check and write, and hence highly error-prone. The aim of the work is hence to automate the process and verify the correctness of these self-stabilizing rendezvous algorithms using model checking.

Model checking relies on the definition of a verification model consisting exclusively of finite values. In particular, the domain of variables must be kept to a minimum. In the case of rendezvous of robots with lights, the number of robots and the number of colors are finite and very small. However, the position of the robots and the time of their activations have all infinite domains. This means that the verification model must capture the important characteristics of the original model while representing everything in a tractable state space. In other words, the only way to obtain automated proofs of correctness in the continuous space context through model checking is to use a more abstract verification model.

Methodology

The approach consists of two parts: (a) the definition of the verification model and its correctness, and (b) the implementation of the verification model and algorithms in the model-checker.

First, we prove several important results that are used to support the verification model. In particular, there is obviously an inherent loss of generality when representing infinite domains with finite values. In this case, our verification model is designed to fail conservatively, in the sense that an incorrect algorithm must always result in a failed verification, whereas a correct algorithm may not always result in a successful one. This point is crucial because this is why this approach is sound.

The key aspect of the verification model consists in representing the Euclidean environment with only two discrete values (or three depending on assumptions) with an appropriate way to resolve movements. The difficult and subtle part is obviously the latter one and is presented in details in the full version of this manuscript [2].

Second, we express the verification model in the SPIN model checker [6]. Given a model expressed in the Promela language (a concurrent language reminiscent of Hoare's CSP) and a linear temporal logic formula, the model checker explores every possible paths of the model until it reaches an invalid state (one that violates the formula) from an initial state, in which case it reports that path as a counter-example execution that violates the temporal logic formula. Otherwise, it reports success. Since the model checker performs an *exhaustive search*, branching out at every non-deterministic choice, the condition is then proved to hold. Model checking has gone a long way since the 70s when it began, and current solvers are now able to handle millions of states without much problem.

We wrote the Promela model to be as modular as possible. In particular, the same framework is used to model seven different synchrony models, including FSYNC, SSYNC, and ASYNC. The rendezvous algorithms are expressed as functions mapping an observation to a color and a move. Based on a theorem proved in the first part, a fair execution can be modeled by limiting the number of consecutive activations by a constant value.

Table 1 Results of model-checking liveness of some known algorithms.

Central.	FSYNC	SSYNC	LC-atom. ASYNC	Move-atom. ASYNC	ASYNC	
-	-	-	-	-	-	Neither robots move
-	✓	-	-	-	-	Move to midpoint
✓	-	-	-	-	-	Move to other
✓	✓	✓	✓	-	-	Viglietta [10] 2 colors
✓	✓	✓	✓	✓	✓	Viglietta [10] 3 colors
✓	✓	✓	✓	✓	✓	Heriban+ [5] 2 colors
✓	✓	✓	-	-	-	Flocchini+ [4] 3 colors
✓	✓	✓	✓	-	-	Okumura+ [7] 5 colors
✓	-	-	-	-	-	Okumura+ [7] 4 col.; quasi-SS
✓	-	-	-	-	-	Okumura+ [7] 3 col.; non-SS

Validation

To assess the verification model and its SPIN implementation, we have checked ten different rendezvous algorithms: three trivial baseline algorithms as well as seven known algorithms from the literature. For each algorithms, two of which are not self-stabilizing, it is widely-known in which models they achieve rendezvous or fail. Table 1 summarizes the results (either positive or negative with a counter-example) obtained from model-checking these algorithms in six different synchrony models, including the three most common (FSYNC, SSYNC, and ASYNC). Each outcome of the resulting 60 tests is consistent with the literature, and the entire test runs in a few minutes on a regular laptop.

References

1. S. Das, P. Flocchini, G. Prencipe, N. Santoro, and M. Yamashita. Autonomous mobile robots with lights. *Theor. Comput. Sci.*, 609:171–184, 2016. doi:10.1016/j.tcs.2015.09.018.
2. X. Défago, A. Heriban, S. Tixeuil, and K. Wada. Using Model Checking to Formally Verify Rendezvous Algorithms for Robots with Lights in Euclidean Space. CoRR abs/1907.09871, arXiv, July 2019. arXiv:1907.09871.
3. P. Flocchini, G. Prencipe, N. Santoro, and P. Widmayer. Gathering of asynchronous robots with limited visibility. *Theor. Comput. Sci.*, 337(1-3):147–168, 2005. doi:10.1016/j.tcs.2005.01.001.
4. P. Flocchini, N. Santoro, G. Viglietta, and M. Yamashita. Rendezvous with constant memory. *Theor. Comput. Sci.*, 621:57–72, 2016. doi:10.1016/j.tcs.2016.01.025.
5. A. Heriban, X. Défago, and S. Tixeuil. Optimally Gathering Two Robots. In *Proc. ICDCN*, pages 3:1–3:10, January 2018. doi:10.1145/3154273.3154323.
6. G. Holzmann. *The SPIN Model Checker: Primer & Reference Manual.* Addison-Wesley, 2004.
7. T. Okumura, K. Wada, and X. Défago. Optimal Rendezvous $\mathcal{L}$-Algorithms for Asynchronous Mobile Robots with External-Lights. In *Proc. OPODIS*, pages 24:1–16, December 2018. doi:10.4230/LIPIcs.OPODIS.2018.24.
8. T. Okumura, K. Wada, and Y. Katayama. Brief Announcement: Optimal Asynchronous Rendezvous for Mobile Robots with Lights. In *Proc. SSS*, pages 484–488, November 2017. doi:10.1007/978-3-319-69084-1_36.
9. I. Suzuki and M. Yamashita. Distributed Anonymous Mobile Robots: Formation of Geometric Patterns. *SIAM J. Comput.*, 28(4):1347–1363, 1999. doi:10.1137/S009753979628292X.
10. G. Viglietta. Rendezvous of Two Robots with Visible Bits. In *Proc. ALGOSENSORS*, pages 291–306, September 2013. doi:10.1007/978-3-642-45346-5_21.

Brief Announcement: Massively Parallel Approximate Distance Sketches

Michael Dinitz
Johns Hopkins University, Baltimore, MD, United States
mdinitz@cs.jhu.edu

Yasamin Nazari
Johns Hopkins University, Baltimore, MD, United States
ynazari@jhu.edu

Abstract

Data structures that allow efficient distance estimation have been extensively studied both in centralized models and classical distributed models. We initiate their study in newer (and arguably more realistic) models of distributed computation: the Congested Clique model and the Massively Parallel Computation (MPC) model. In MPC we give two main results: an algorithm that constructs stretch/space optimal distance sketches but takes a (small) polynomial number of rounds, and an algorithm that constructs distance sketches with worse stretch but that only takes polylogarithmic rounds. Along the way, we show that other useful combinatorial structures can also be computed in MPC. In particular, one key component we use is an MPC construction of the hopsets of [2]. This result has additional applications such as the first polylogarithmic time algorithm for constant approximate single-source shortest paths for weighted graphs in the low memory MPC setting.

2012 ACM Subject Classification Theory of computation → Massively parallel algorithms

Keywords and phrases Distance Sketches, Massively Parallel Computation, Congested Clique

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.42

Related Version The full version of this paper is available at `https://arxiv.org/abs/1810.09027`.

Funding Supported in part by NSF awards CCF-1464239 and CCF-1535887.

1 Introduction

A common task when performing graph analytics is to compute distances between vertices. This has motivated the study of shortest path algorithms in essentially every interesting model of computation. We focus on two models which correspond to modern big-data graph analytics: Congested Clique [6] and Massively Parallel Computation (MPC) [4]. The MPC model in particular has recently received significant attention, as it captures many modern data analytics frameworks such as MapReduce, Hadoop, and Spark. Since these are important models of distributed storage and computation, and computing distances in graphs is an important primitive, we have an obvious question: in MPC or Congested Clique, can we compute distances between nodes sufficiently quickly to support important graph analytics? While one side effect of our techniques is indeed a state of the art algorithm for shortest paths in MPC, the focus of this paper is on getting around the limitations of these models by allowing preprocessing of the (distributed) graph. We will build a data structure known as *approximate distance sketches*, which will then let us (approximately) answer any distance query using at most two rounds of network communication. So our focus is on how to compute these data structures efficiently, since once they are computed distance estimates become fast and easy. We show that in both the Congested Clique and the MPC models, we can compute oracles/sketches which essentially match the best centralized bounds in time that is only a small polynomial. In MPC, we can go even further and compute slightly suboptimal sketches in time that is only polylogarithmic.

© Michael Dinitz and Yasamin Nazari;
licensed under Creative Commons License CC-BY
33rd International Symposium on Distributed Computing (DISC 2019).
Editor: Jukka Suomela; Article No. 42; pp. 42:1–42:3

Distance Oracles and Sketches. Even in many centralized applications, the time it takes to compute exact distances in graphs is undesriable, and similarly the memory that it would take to store all $\binom{n}{2}$ distances is also undesirable. This motivated Thorup and Zwick [5] to propose a space-efficient data structure which can quickly report an approximation of the true distance for any pair of vertices. In other words, by spending some time up front to compute this data structure, any algorithm used in the future can quickly obtain provably accurate distance estimates. More formally, an approximate distance oracle is said to have *stretch* t if, when queried on $u, v \in V$, it returns a value $d'(u, v)$ such that $d(u, v) \leq d'(u, v) \leq t \cdot d(u, v)$ for all $u, v \in V$, where $d(u, v)$ denotes the shortest-path distance between u and v. For any constant k, Thorup and Zwick's (centralized) construction has expected size $O(kn^{1+1/k})$, stretch $(2k-1)$, query time $O(k)$, and preprocessing time $O(kmn^{1/k})$. Also, this data structure can be "broken up" into n pieces, each of size $O(kn^{1/k} \log n)$, so that the estimate $d'(u, v)$ can be computed just from the piece for u and the piece for v. These are called *distance sketches*.

Model. Our main focus is the *Massively Parallel Computation*, or MPC model. In this model there is an input of size N which is arbitrarily distributed over N/S machines, each of which has $S = N^{\epsilon}$ memory for some $0 < \epsilon < 1$. Every machine can communicate with every other machine in the network, but each machine in each round can have total I/O of at most S. Specifically, for graph problems the total memory N is $O(|E|)$ words. The low memory setting is the more challenging (but arguably more realistic) setting in which each machine has $O(n^{\gamma}), \gamma < 1$ memory, which we denote by MPC(n^{γ}).

2 Our Results

We initiate the study of distance sketches in the MPC model. Our techniques also extends to the Congested Clique and streaming models. Exact results can be found in the full paper. We first show that distance sketches with the same guarantees as the centralized Thorup-Zwick distance oracles can be implemented in MPC, but with a polynomial (sublinear) round complexity. Since such a high round complexity is generally considered impractical, so we also give a different (but related) algorithm which achieves polylogarithmic round complexity at the price of larger stretch. More formally,

▶ **Theorem 1.** *Consider a graph $G = (V, E)$ where $m = \Omega(kn^{1+1/k} \log n)$, for some integer $k \geq 2$. Then there is an algorithm in MPC(n^{γ}) (with $0 < \gamma < 1$) that constructs Thorup-Zwick distance sketches with stretch $O(k^2)$ and size $O(kn^{1/k} \log n)$ and with high probability completes in $O(\frac{k}{\gamma} \cdot (\log^3 n \cdot \log^3 k)^{2 \log k})$ rounds.*

As a side effect of our techniques, we immediately get an algorithm for computing approximate single-source shortest paths (SSSP). We show that we can compute an $O(1)$-approximation in only polylogarithmic time under a certain assumption on the density of the input graph.

3 Techniques

Our main approach is to combine constructions of *hopsets* with efficient distributed constructions of Thorup-Zwick distance oracles/sketches. In particular, Das Sarma et al. [1] showed that Thorup-Zwick sketches could be computed in the CONGEST model, but the time depended on the graph diameter. Roughly speaking, we use hopsets to reduce the diameter of

the graph while preserving distances by adding in a carefully chosen set of weighted "shortcut" edges. We use a hopset construction proposed by Elkin and Neiman [2]. To implement their algorithm in the MPC model, we need to handle some technical difficulties particularly when the space per machine is $o(n)$. Not surprisingly, both [1] and [2] use as a fundamental primitive a "restricted" version of the classical Bellman-Ford shortest-path algorithm that ends early. Hence the first step for us is implementing this restricted Bellman-Ford in the MPC model. When implementing restricted Bellman-Ford in low-memory MPC, the main difficulty is that since the memory at each server is $o(n)$, a single server cannot "simulate" a node in Bellman-Ford. It takes many machines to store the edges incident on any particular node. We first show that it is possible to implement Bellman-Ford in low-memory MPC with very little additional overhead. Once we develop this tool, we argue that the hopsets of [2]) can be constructed in MPC. Our implementation of Bellman-Ford and this hopset construction, as well as a few other primitives we develop for low-memory MPC (e.g., finding minimum or broadcasting on a range of machines), may be of independent interest.

Directly implementing the hopset algorithm of [2] requires a polynomial number of rounds to obtain polylogarithmic hopbound. Even after using hopsets, we would still need polynomial time to construct constant stretch distance sketches. We overcome this issue and improve the running time using two ideas. First, we show that by relaxing the model to allow small additional total memory (either through extra space per machine or additional machines), we can run our algorithms in polylogarithmic number of rounds. In other words, the MPC model is very delicate: a small polynomial amount of extra space allows us to decrease running times not just by that polynomial, but from polynomial to polylogarithmic. So we just need to argue that there is a way of obtaining extra memory without actually changing the model assumptions. This is our second idea: by constructing a spanner we can sparsify the graph while keeping the memory per machine and number of machines the same. Thus from the perspective of the spanner, it will appear that we do indeed have "extra" memory. The idea of sparsifying the input to obtain extra resources has already proved to be powerful in related contexts (for example, [3] recently used spanners to give a work-efficient PRAM metric embedding algorithm). To the best of our knowledge, though, this idea has not yet appeared in the MPC graph algorithms literature.

References

1 M. Dinitz A. Sarma and G. Pandurangan. Efficient distributed computation of distance sketches in networks. *Distributed Computing*, 2015.
2 M. Elkin and O. Neiman. Hopsets with constant hopbound, and applications to approximate shortest paths. In *FOCS*, 2016.
3 S. Friedrichs and C. Lenzen. Parallel metric tree embedding based on an algebraic view on moore-bellman-ford. *Journal of the ACM (JACM)*, 2018.
4 P. Koutris P. Beame and D. Suciu. Communication steps for parallel query processing. In *PODS*, 2013.
5 M. Thorup and U. Zwick. Approximate distance oracles. *Journal of the ACM (JACM)*, 2005.
6 E. Pavlov Z. Lotker, B. Patt-Shamir and D. Peleg. Minimum-weight spanning tree construction in O (log log n) communication rounds. *SIAM Journal on Computing*, 2005.

Brief Announcement: Neighborhood Mutual Remainder and Its Self-Stabilizing Implementation of Look-Compute-Move Robots

Shlomi Dolev
Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, Israel
dolev@cs.bgu.ac.il

Sayaka Kamei
Graduate School of Engineering, Hiroshima University, Japan
s-kamei@se.hiroshima-u.ac.jp

Yoshiaki Katayama
Graduate School of Engineering, Nagoya Institute of Technology, Japan
katayama@nitech.ac.jp

Fukuhito Ooshita
Graduate School of Science and Technology, Nara Institute of Science and Technology, Japan
f-oosita@is.naist.jp

Koichi Wada
Faculty of Science and Engineering, Hosei University, Japan
wada@hosei.ac.jp

Abstract

In this paper, we define a new concept *neighborhood mutual remainder* (NMR). An NMR distributed algorithms should satisfy *global fairness*, *l-exclusion* and *repeated local rendezvous* requirements. We give a simple self-stabilizing algorithm to demonstrate the design paradigm to achieve NMR, and also present applications of NMR to a *Look-Compute-Move* robot system.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases neighborhood mutual remainder, self-stabilization, LCM robot

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.43

Related Version `https://arxiv.org/abs/1903.02843`

Funding This work was supported in part by JSPS KAKENHI No. 17K00019, 18K11167, 19K11823 and 19K11828, Rita Altura Trust Chair in Computer Science, the Ministry of Science and Technology, Israel &Japan Science and Technology Agency (JST) SICORP (Grant#JPMJSC1806) and the German Research Funding Organization (DFG, Grant#8767581199).

1 Introduction

Distributed systems sometimes encounter *mutually exclusive* operations such that, while one operation is executed by a participant, another operation cannot be executed by the participant and its neighboring participants. For example, we can consider a Look-Compute-Move (LCM) robot system, where each robot repeats executing cycles of look, compute, and move phases. Some algorithms assume the move-atomic property, that is, while robot r executes look and compute phases, r's neighbors cannot execute a move phase. In this case, the move operation and the look/compute operations are mutually exclusive.

To execute mutually exclusive operations consistently, participants should schedule the operations carefully. One may think we can apply *mutual exclusion* or *local mutual exclusion* to solve the local synchronization problem. Mutual exclusion (resp., local mutual exclusion)

© Shlomi Dolev, Sayaka Kamei, Yoshiaki Katayama, Fukuhito Ooshita, and Koichi Wada;
licensed under Creative Commons License CC-BY
33rd International Symposium on Distributed Computing (DISC 2019).
Editor: Jukka Suomela; Article No. 43; pp. 43:1–43:3

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

guarantees that no two participants (resp., no two neighboring participants) enter a critical section (CS) at the same time. Indeed, if participants execute mutually exclusive operations only when they are in the CS, they can keep the consistency because no two neighboring participants execute the mutually exclusive operations at the same time. On the other hand, this approach seems expensive because participants execute the operations sequentially despite that they are allowed to execute the same operation simultaneously. Also, to realize local mutual exclusion, participants should achieve symmetry breaking because one participant should be selected to enter the CS. However, in highly-symmetric distributed systems such as the LCM robot system, it is difficult or even impossible to achieve deterministic symmetry breaking and thus achieve local mutual exclusion.

From this motivation, we define the *neighborhood mutual remainder* (NMR) distributed task over a distributed system with a general, non-necessarily complete, communication graph. An NMR distributed algorithm should satisfy *global fairness*, *l-exclusion*, and *repeated local rendezvous* requirements. Global fairness is satisfied when each participant executes the CS infinitely often, l-exclusion is satisfied when at most l neighboring processes enter the CS at the same time, and repeated local rendezvous is satisfied when, for each participant, infinitely often no participant in the closed neighborhood is in the critical or trying sections. Unlike the classical (local) mutual exclusion problem, the NMR allows up to l neighboring participants to be simultaneously in the CS, but requires a guarantee for neighborhood rendezvous in the remainder.

For example, in the LCM robot system, the move-atomic property can be achieved by NMR: Each robot executes look and compute phases when it is in the CS, and executes a move phase only when no robot in its closed neighborhood is in the CS. While some robot executes look and compute phases, none of its neighbors executes a move phase. From the global fairness and local rendezvous properties, all robots execute look, compute, and move phases infinitely often.

Our contributions. In this BA, we formalize the concept of NMR, and give a design paradigm to achieve NMR. To demonstrate the design paradigm, we consider synchronous distributed systems and give a simple self-stabilizing algorithm for NMR. To consider the simplest case, we assume $l = \Delta + 1$, where Δ is the maximum degree, that is, l-exclusion is always satisfied. In this case, the NMR does not require symmetry breaking, however, is still useful for some applications.

In the full version [1], to demonstrate applicability of NMR, we implement a self-stabilizing synchronization algorithm for the LCM robot system by using the aforementioned design paradigm. First, we realize the move-atomic property in a self-stabilizing manner on the assumption that robots repeatedly receive clock pulses at the same time. After that, we extend it to the assumption that robots receive clock pulses at different times but the duration between two pulses is identical for all robots. Lastly, on the same assumption, we implement the fully synchronous (FSYNC) model in a self-stabilizing manner. This research presents the first self-stabilizing implementation of the LCM synchronization, allowing the implementation in practice of any self-stabilizing or stateless robot algorithm, where robots possess independent clocks that are advanced in the same speed.

2 Preliminary

A distributed system is represented by an undirected connected graph $G = (V, E)$, where $V = \{v_0, \ldots, v_{k-1}\}$ is a set of processes and $E \subseteq V \times V$ is a set of communication links between processes. Processes are anonymous and identical. Process v_i is a neighbor of v_j if

Algorithm 1 Self-Stabilizing NMR Algorithm for $l = \Delta + 1$. Pseudo-Code for v_i.

1: **Upon** a global pulse
2: $N_i := |N[i]|$; $MaxN_i := \max\{N_j \mid v_j \in N[i]\}$; $Clock_i := (Clock_i + 1) \mod (MaxN_i + 1)$;
3: **if** $Clock_i = 1$ **then** enter the critical section and leave before the next pulse;
4: **else if** $\forall v_j \in N[i][Clock_j \neq 1]$ **then** rendezvous in the remainder section;

$(v_i, v_j) \in E$ holds. A neighborhood of v_i is denoted by $N(i) = \{v_j \mid (v_i, v_j) \in E\}$, and the degree of v_i is denoted by $\delta(i) = |N(i)|$. Let $\Delta = \max\{\delta(i) \mid v_i \in V\}$. A closed neighborhood of v_i is denoted by $N[i] = N(i) \cup \{v_i\}$.

Each process is a state machine that changes its state by actions. We adopt the *state-reading model* as a communication model. In this model, each v_i can directly read states of all processes in $N[i]$ simultaneously and update its own state. Processes operate synchronously based on global pulses. That is, all processes regularly receive the pulse at the same time, and operate when they receive the pulse. The duration of local computation is sufficiently small so that every process completes the local computation before the next pulse.

▶ **Definition 1** (Neighborhood mutual remainder (NMR)). *The system achieves NMR if the following three properties hold.*

- *Global fairness: Every process infinitely often enters the CS.*
- *l-exclusion: For every process v_i, at most l processes in $N[i]$ enter the CS simultaneously.*
- *Repeated local rendezvous: For every process v_i, infinitely many instants exist such that no process in $N[i]$ is in the critical or trying section.*

3 A self-stabilizing algorithm for neighborhood mutual remainder

In this section, we give a design paradigm to achieve NMR. As an example, we realize a self-stabilizing algorithm to achieve NMR in case of $l = \Delta + 1$. We say an algorithm is self-stabilizing if, from any initial configuration, the system eventually exhibits the desired behavior.

The self-stabilizing NMR algorithm is given in Algorithm 1. Let us consider a simple setting where $|N[i]|$ is identical for any v_i. Every process v_i maintains a clock $Clock_i$ that is incremented by 1 modulo $(|N[i]| + 1)$ in every pulse. The value of $Clock_i$ may differ from the value of $Clock_j$, for a neighbor v_j of v_i. Say, for the sake of simplicity, that v_i may possess the CS only when $Clock_i = 1$. Thus, ensuring that there is a configuration in which all processes in the remainder is equivalent to ensuring that there is a configuration in which the values of all the above clocks are not equal to 1. Using the pigeon-hole principle in every $|N[i]| + 1$ consequence pulse clocks, there must be a configuration in which no clock value of neighboring processes is 1 and at the same time $Clock_i$ is not 1 either. Hence, the NMR must hold. Since $|N[i]| \neq |N[j]|$ may hold for some v_i and v_j, we use $MaxN_i = \max\{|N[j]| \mid v_j \in N[i]\}$ instead of $|N[i]|$. Since every process $v_j \in N[i]$ enters the CS at most once in $MaxN_i + 1$ consecutive pulses, we can still use the pigeon-hole principle and hence the NMR must hold.

References

1 S. Dolev, S. Kamei, Y. Katayama, F. Ooshita, and K. Wada. Neighborhood Mutual Remainder: Self-Stabilizing Implementation of Look Compute Move. *arXiv*, 2019. arXiv:1903.02843.

Brief Announcement: Revisiting Consensus Protocols through Wait-Free Parallelization

Suyash Gupta
Exploratory Systems Lab, Department of Computer Science,
University of California, Davis, CA, USA
sugupta@ucdavis.edu

Jelle Hellings
Exploratory Systems Lab, Department of Computer Science,
University of California, Davis, CA, USA
jhellings@ucdavis.edu

Mohammad Sadoghi
Exploratory Systems Lab, Department of Computer Science,
University of California, Davis, CA, USA
msadoghi@ucdavis.edu

Abstract

In this brief announcement, we propose a *protocol-agnostic* approach to improve the design of primary-backup consensus protocols. At the core of our approach is a novel wait-free design of running several instances of the underlying consensus protocol *in parallel*. To yield a high-performance parallelized design, we present coordination-free techniques to order operations across parallel instances, deal with instance failures, and assign clients to specific instances. Consequently, the design we present is able to reduce the load on individual instances and primaries, while also reducing the adverse effects of any malicious replicas. Our design is fine-tuned such that the instances coordinated by non-faulty replicas are *wait-free*: they can continuously make consensus decisions, independent of the behavior of any other instances.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Consensus, primary-backup, high-performance, wait-free parallelization

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.44

Related Version The technical report on which this brief announcement is based is available at `https://arxiv.org/abs/1908.01458`.

1 Introduction

At the core of any blockchain application is a *consensus protocol* that facilitates replicating data across a group of servers (or replicas), some of which can fail or act malicious [5]. Commonly, these protocols use the *primary-backup model* pioneered in the Practical Byzantine Fault Tolerant consensus protocol [2]. In these BFT-style protocols, a single replica is designated as *the primary* and is responsible for coordinating the consensus decisions, while all the other replicas perform the *backup role*. The primary-backup model simplifies the development of consensus protocols substantially: when a primary is non-malicious, then even the simplest broadcast replication protocols suffice. However, the only complication in these consensus protocols is in the way they deal with malicious primaries: malicious behavior must be either detected (after which the primary can be replaced) or prevented altogether. This simplicity of the primary-backup model negatively affects its performance in three ways [1, 3]:

1. *Primary load.* The primary not only has to perform the primary tasks, but also the backup role (as it is itself a replica). Consequently, the primary receives a higher load than other replicas, and this load at the primary can become a *bottleneck in the overall system*

© Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi;
licensed under Creative Commons License CC-BY

throughput. This is especially the case in fine-tuned high-performance consensus protocols that employ complex cryptographic primitive, for example, to reduce communication overheads or to improve resilience.

2. *Primary replacement*. A primary-backup consensus protocol works only when the primary behaves in accordance with the protocol. If the primary acts malicious or is faulty, then it will be replaced. However, detection of such behaviors requires setting timers. Further, replacing a faulty primary usually takes a while. During this time the system is unable to handle requests, which negatively affects its overall *throughput*.
3. *Malicious behavior*. Primary-backup consensus protocols are only capable of detecting catastrophic failures that prevent new consensus decisions altogether, but they fail to detect or deal with primaries that *affect the performance of the system in other ways*, for example, a malicious primary could reduce or throttle the throughput of the system.

To the best of our knowledge, no approach is yet able to address all these limitations of primary-backup consensus protocols. In this brief announcement, we address these limitations in a *protocol-agnostic* manner by exploiting parallelization.[1] In our *paradigm*, we run several instances of the underlying consensus protocol *in parallel* and balance the system load among these parallel instances. This parallelism helps to reduce the *load per primary* and mitigates the negative impacts of a single primary on the *throughput of the system*.

2 Parallelizing consensus

Consider a system with $\mathbf{n}$ replicas, of which $\mathbf{f}$ are faulty. At the core of our parallelization paradigm is the coordination of $\mathbf{m}$, $\mathbf{f} < \mathbf{m} \leq \mathbf{n}$, instances of an off-the-shelf primary-backup consensus protocol running *in parallel*. This implies that a single round of our paradigm coordinates *multiple parallel consensus rounds*, each of which is initiated and managed by a *distinct* primary $\mathcal{P}_i$ for the instance $\mathcal{I}_i$, $1 \leq i \leq \mathbf{m}$. Each consensus decision succeeds whenever $\mathcal{P}_i$ is non-faulty. This approach to parallelization raises several important challenges:

1. For optimal throughput, we need to ensure that each instance is making a *distinct consensus decision*, that is, each instance is processing a distinct client request.
2. Every non-faulty replica should execute all the accepted client requests in *the same order*.
3. When several instances fail in a round and want to *transfer control to new primaries*, then all the non-faulty replicas need to do so in the same manner.

In our design, we address each of these challenges. Figure 1 sketches a high-level overview of a *parallelized consensus round* at replica R. Our paradigm also identifies various ways in which malicious replicas can prevent it from efficently operation. Next, we highlight the design decisions taken to addresses these possible attacks.

Deterministic round execution. The correctness of the underlying consensus protocol, used by instances $\mathcal{I}_1, \ldots, \mathcal{I}_{\mathbf{m}}$, can guarantee that each non-faulty replica derives the same set of client requests in round ρ. Hence, our paradigm behaves correctly whenever all non-faulty replicas can determine the same order of execution of these client requests. To avoid that malicious replicas can influence, predict, or reliably exploit the order of execution to their advantage, we permute the order of execution based on the set of client requests accepted in round ρ (some of which are proposed by non-faulty replicas).

[1] A complementary approach to increase parallelism is via partial replication through sharding [4].

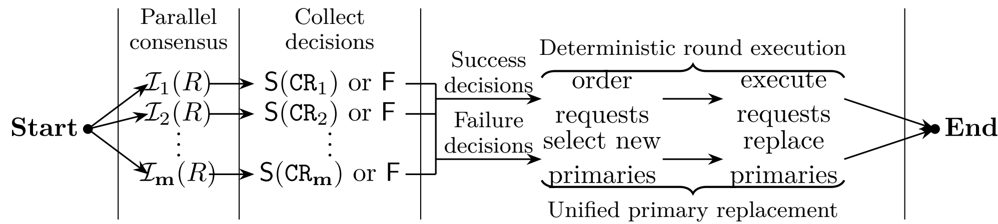


Figure 1 A high-level overview of a replica R. The replica coordinates a single consensus round among $\mathbf{m}$ instances of some *consensus protocol*. Each instance yields a consensus decision. The success decisions $\mathsf{S}(\cdot)$ yield a set of client requests, which are executed in a deterministic order. The failure decisions F are collected and can be used to replace primaries in a unified manner.

Dealing with primary failure. In primary-backup consensus protocols, the primary can fail or act malicious. Our paradigm supports two ways to deal with such failures. The straightforward approach is to suspend instances with failed primaries and to try recovering these failed primaries after some delay. To avoid recovering too often, this delay is doubled after each failure. We also support replacement of failed primaries by other replicas. To do so, we provide a *unified primary replacement* protocol that only requires coordination among the instances with failed primaries.

Consistent handling of client requests. To ensure that subsequent client requests are always processed in order, we assign clients to instances in a round-robin manner. We do this by requiring each instance $\mathcal{I}_i$, $1 \leq i \leq \mathbf{m}$, to only deal with client requests of a client c if $i = c \bmod \mathbf{m}$. We notice that a client c can be assigned to an instance with a faulty primary that might ignore its request. To deal with this situation, we allow clients to periodically issue instance-change requests to reassign them to other instances. To assure balanced load among the instances, a non-faulty instance only accepts an instance-change request if it has not been yet assigned $\lceil \mathbf{c}/(\mathbf{n} - \mathbf{f}) \rceil$ clients, where $\mathbf{c}$ refers to the total number of clients.

Wait-free parallelization. To ensure the correctness of our parallelization paradigm, we do not require any non-faulty instance to wait for other instances. In our paradigm, the *execution of client requests* in round ρ has no influence on the consensus decisions of future rounds. Second, the instances arriving at successful consensus decisions do not require any coordination. The only required coordination is between instances with failed primaries. Hence, instances that arrived at successful consensus decisions in the current round are free to make consensus decisions for the future rounds, while the *execution* of the client requests of previous rounds occurs in the background. Furthermore, we have coordination-free ways to detect and sanction malicious behavior of primaries (e.g., throttling performance). Combined, these approaches guarantee that instances with non-faulty primaries are *wait-free*: they are always able to operate at maximum throughput and will always see their client requests executed within bounded time, this independent of any malicious behavior in other instances.

References

1 Pierre-Louis Aublin, Sonia Ben Mokhtar, and Vivien Quéma. RBFT: Redundant byzantine fault tolerance. In *ICDCS*, 2013.
2 Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *OSDI*, 1999.
3 Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. In *NSDI*, 2009.
4 Jelle Hellings and Mohammad Sadoghi. Brief Announcement: The Fault-Tolerant Cluster-Sending Problem. In *33nd International Symposium on Distributed Computing (DISC 2019)*, volume 146 of *LIPIcs*, pages 45:1–45:3, 2019. doi:10.4230/LIPIcs.DISC.2019.45.
5 Faisal Nawab and Mohammad Sadoghi. Blockplane: A Global-Scale Byzantizing Middleware. In *ICDE*, 2019.

Brief Announcement: The Fault-Tolerant Cluster-Sending Problem

Jelle Hellings
Exploratory Systems Lab, Department of Computer Science,
University of California, Davis, CA, USA
jhellings@ucdavis.edu

Mohammad Sadoghi
Exploratory Systems Lab, Department of Computer Science,
University of California, Davis, CA, USA
msadoghi@ucdavis.edu

Abstract

The development of fault-tolerant distributed systems that can tolerate Byzantine behavior has traditionally been focused on consensus protocols, which support fully-replicated designs. For the development of more sophisticated high-performance Byzantine distributed systems, more specialized fault-tolerant communication primitives are necessary, however.

In this brief announcement, we identify the *cluster-sending problem* – the problem of sending a message from one Byzantine cluster to another Byzantine cluster in a reliable manner – as such an essential communication primitive. We not only formalize this fundamental problem, but also establish lower bounds on the complexity of this problem under crash failures and Byzantine failures. Furthermore, we develop practical cluster-sending protocols that meet these lower bounds and, hence, have optimal complexity. As such, our work provides a strong foundation for the further exploration of novel designs that address challenges encountered in fault-tolerant distributed systems.

2012 ACM Subject Classification Theory of computation → Communication complexity; Theory of computation → Distributed algorithms

Keywords and phrases Byzantine clusters, message sending, lower bound, optimal protocol

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.45

Related Version The technical report on which this brief announcement is based is available at `https://arxiv.org/abs/1908.01455`.

1 Introduction

Recently, the emergence of *blockchain technology* has fueled a renewed interest in the development of fault-tolerant distributed systems. The main focus of current developments is mostly limited to fully-replicated systems in which each participating replica has the same role. We envision the design and development of more sophisticated high-performance Byzantine systems in which replicas have specialized roles. An example of such a system would be a *sharded geo-scale design* in which data is kept in *local Byzantine clusters*. In such a sharded geo-scale design, many queries can efficiently be answered by involving only a single cluster [1, 2]. In this way, a sharded design will often improve scalability when dealing with massive large-scale databases. For answering more complex queries, we need cooperation between different clusters, however. Hence, to enable the design and development of such systems, we need reliable ways for Byzantine clusters to communicate and cooperate. We believe that the existing consensus protocols are insufficient to fulfill this aim: we can run a single global consensus protocol among all replicas in all clusters to enable sharing of data and queries, but this would be at high – *quadratic* – communication costs for all replicas involved and would eliminate any possible scaling benefits of a clustered design. Indeed, we believe that there is a pressing need for more specialized Byzantine communication primitives. In this announcement we formalize and study one such primitive, the *cluster-sending problem*.

© Jelle Hellings and Mohammad Sadoghi;
licensed under Creative Commons License CC-BY
33rd International Symposium on Distributed Computing (DISC 2019).
Editor: Jukka Suomela; Article No. 45; pp. 45:1–45:3

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2 On the Fault-Tolerant Cluster-sending Problem

A *cluster* $\mathcal{C}$ is a set of replicas. We write $\mathsf{f}(\mathcal{C}) \subseteq \mathcal{C}$ to denote the set of *faulty replicas* in $\mathcal{C}$ and $\mathsf{nf}(\mathcal{C}) = \mathcal{C} \setminus \mathsf{f}(\mathcal{C})$ to denote the set of *non-faulty replicas* in $\mathcal{C}$. We write $\mathbf{n}_{\mathcal{C}} = |\mathcal{C}|$, $\mathbf{f}_{\mathcal{C}} = |\mathsf{f}(\mathcal{C})|$, and $\mathbf{nf}_{\mathcal{C}} = |\mathsf{nf}(\mathcal{C})|$ to denote the number of replicas, faulty replicas, and non-faulty replicas in the cluster, respectively. We extend the notations $\mathsf{f}(\cdot)$, $\mathsf{nf}(\cdot)$, $\mathbf{n}_{(\cdot)}$, $\mathbf{f}_{(\cdot)}$, and $\mathbf{nf}_{(\cdot)}$ to arbitrary sets of replicas. A *cluster system* $\mathfrak{S}$ is a finite set of clusters such that communication between replicas in a cluster is *local* and communication between clusters is *non-local*. We assume that there is no practical bound on local communication, while global communication is limited, costly, and to be avoided. If $\mathcal{C}_1, \mathcal{C}_2 \in \mathfrak{S}$ are distinct clusters, then we assume that $\mathcal{C}_1 \cap \mathcal{C}_2 = \emptyset$: no replica is part of two distinct clusters.

▶ **Definition 1.** *Let $\mathfrak{S}$ be a system and $\mathcal{C}_1, \mathcal{C}_2 \in \mathfrak{S}$ be two clusters with non-faulty replicas ($\mathsf{nf}(\mathcal{C}_1) \neq \emptyset$ and $\mathsf{nf}(\mathcal{C}_2) \neq \emptyset$). The* cluster-sending problem *is the problem of sending a value v from $\mathcal{C}_1$ to $\mathcal{C}_2$ such that:* **(1)** *all non-faulty replicas in $\mathcal{C}_2$* receive *the value v;* **(2)** *only if all non-faulty replicas in $\mathcal{C}_1$* agree *upon sending the value v to $\mathcal{C}_2$ will non-faulty replicas in $\mathcal{C}_2$ receive v; and* **(3)** *all non-faulty replicas in $\mathcal{C}_1$ can* confirm *that the value v was received.*

In this announcement, we use the notation $i \operatorname{sgn} j$, with $i, j \geq 0$ and sgn the sign function, to denote i if $j > 0$ and 0 otherwise.

Lower bounds for the cluster-sending problem. First, we consider systems with only *crash failures*, in which case we can lower bound the number of messages exchanged. This lower bound is entirely determined by the maximum number of messages that can get *lost* due to faulty replicas not sending messages or ignoring received messages. In situations in which some replicas need to send or receive *multiple* messages, the capabilities of faulty replicas to ignore messages is likewise multiplied. E.g., when the number of senders outnumbers the receivers, then some receivers must receive multiple messages. As these receivers could be faulty, this means they could cause loss of multiple messages. By a thorough analysis, we end up with the following lower bounds:

▶ **Theorem 2.** *Let $\mathfrak{S}$ be a system with crash failures, let $\mathcal{C}_1, \mathcal{C}_2 \in \mathfrak{S}$, and let $\{i, j\} = \{1, 2\}$ such that $\mathbf{n}_{\mathcal{C}_i} \geq \mathbf{n}_{\mathcal{C}_j}$. Let $q_i = (\mathbf{f}_{\mathcal{C}_i} + 1) \operatorname{div} \mathbf{nf}_{\mathcal{C}_j}$, let $r_i = (\mathbf{f}_{\mathcal{C}_i} + 1) \bmod \mathbf{nf}_{\mathcal{C}_j}$, and let $\sigma_i = q_i \mathbf{n}_{\mathcal{C}_j} + r_i + \mathbf{f}_{\mathcal{C}_j} \operatorname{sgn} r_i$. Any protocol that solves the cluster-sending problem in which $\mathcal{C}_1$ sends a value v to $\mathcal{C}_2$ needs to exchange at least σ_i messages.*

Next, we look at systems with *Byzantine failures and replica signing* (e.g., using digital signatures and a public-key cryptography infrastructure). In this environment, we prove a lower bound on the number of replica signatures (certificates) exchanged. In this case, the receiving cluster $\mathcal{C}_2$ must eventually receive $\mathbf{f}_{\mathcal{C}_1} + 1$ distinct certificates signed by distinct replicas in $\mathcal{C}_1$. Only after receipt of these $\mathbf{f}_{\mathcal{C}_1} + 1$ certificates can the replicas in $\mathcal{C}_2$ conclude that at least one of these certificates was sent by a non-faulty replica in $\mathcal{C}_1$. A thorough analysis reveals the following lower bounds:

▶ **Theorem 3.** *Let $\mathfrak{S}$ be a system with Byzantine failures and replica signing and let $\mathcal{C}_1, \mathcal{C}_2 \in \mathfrak{S}$. Consider the cluster-sending problem in which $\mathcal{C}_1$ sends a value v to $\mathcal{C}_2$.*

1. *Let $q_1 = (2\mathbf{f}_{\mathcal{C}_1} + 1) \operatorname{div} \mathbf{nf}_{\mathcal{C}_2}$, $r_1 = (2\mathbf{f}_{\mathcal{C}_1} + 1) \bmod \mathbf{nf}_{\mathcal{C}_2}$, and $\tau_1 = q_1 \mathbf{n}_{\mathcal{C}_2} + r_1 + \mathbf{f}_{\mathcal{C}_2} \operatorname{sgn} r_1$. If $\mathbf{n}_{\mathcal{C}_1} \geq \mathbf{n}_{\mathcal{C}_2}$, then any protocol that solves the cluster-sending problem needs to exchange at least τ_1 certificates.*
2. *Let $q_2 = (\mathbf{f}_{\mathcal{C}_2} + 1) \operatorname{div}(\mathbf{nf}_{\mathcal{C}_1} - \mathbf{f}_{\mathcal{C}_1})$, $r_2 = (\mathbf{f}_{\mathcal{C}_2} + 1) \bmod (\mathbf{nf}_{\mathcal{C}_1} - \mathbf{f}_{\mathcal{C}_1})$, and $\tau_2 = q_2 \mathbf{n}_{\mathcal{C}_1} + r_2 + 2\mathbf{f}_{\mathcal{C}_1} \operatorname{sgn} r_2$. If $\mathbf{n}_{\mathcal{C}_2} \geq \mathbf{n}_{\mathcal{C}_1}$, then any protocol that solves the cluster-sending problem needs to exchange at least τ_2 certificates.*

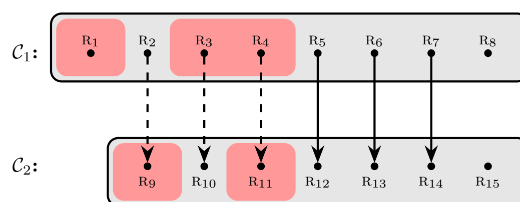


Figure 1 Bijection sending from $\mathcal{C}_1$ to $\mathcal{C}_2$. The faulty replicas are highlighted using a red background. The edges connect replicas $\mathrm{R} \in \mathcal{C}_1$ with $b(\mathrm{R}) \in \mathcal{C}_2$. Each solid edge indicates a message sent and received by non-faulty replicas. Each dashed edge indicates a message sent or received by a faulty replica.

Optimal cluster-sending via partitioned bijective sending. We propose *bijective sending*, a powerful technique that allows the design of highly efficient cluster-sending protocols. In bijective sending, cluster $\mathcal{C}_1$ agrees on a value v. Then, the protocol chooses sets $S_1 \subseteq \mathcal{C}_1$ and $S_2 \subseteq \mathcal{C}_2$ of equal size and instruct each replica in $S_1 \subseteq \mathcal{C}_1$ to send v to a distinct replica in $\mathcal{C}_2$. By choosing S_1 and S_2 sufficiently large, we can guarantee successful cluster-sending. More specific, in the case of a system with *crash failures*, we need to choose $|S_1| = |S_2| = \mathbf{f}_{\mathcal{C}_1} + \mathbf{f}_{\mathcal{C}_2} + 1$. In the case of a system with *Byzantine failures* and replica signing, we need to choose $|S_1| = |S_2| = 2\mathbf{f}_{\mathcal{C}_1} + \mathbf{f}_{\mathcal{C}_2} + 1$. Next, we illustrate bijective sending

▶ **Example 4.** Let $\mathfrak{S}$ be a system with crash failures, let $\mathcal{C}_1 = \{\mathrm{R}_1, \dots, \mathrm{R}_8\} \in \mathfrak{S}$ with $\mathsf{f}(\mathcal{C}_1) = \{\mathrm{R}_1, \mathrm{R}_3, \mathrm{R}_4\}$, and let $\mathcal{C}_2 = \{\mathrm{R}_9, \dots, \mathrm{R}_{15}\} \in \mathfrak{S}$ with $\mathsf{f}(\mathcal{C}_2) = \{\mathrm{R}_9, \mathrm{R}_{11}\}$. We have $\mathbf{f}_{\mathcal{C}_1} + \mathbf{f}_{\mathcal{C}_2} + 1 = 6$. We choose $S_1 = \{\mathrm{R}_2, \dots, \mathrm{R}_7\}$, $S_2 = \{\mathrm{R}_9, \dots, \mathrm{R}_{15}\}$, and $b = \{\mathrm{R}_i \rightarrow \mathrm{R}_{i+7} \mid 2 \leq i \leq 7\}$.

In Figure 1, we sketched this situation. Replica R_2 sends a valid message to R_9. As R_9 is faulty, it might ignore this message. Replicas R_3 and R_4 are faulty and might not send a valid message. Additionally, R_{11} is faulty and might ignore any message it receives. The messages sent from R_5 to R_{12}, from R_6 to R_{13}, and from R_7 to R_{14} are all sent by non-faulty replicas to non-faulty replicas. Hence, these messages all arrive correctly.

The bijective sending techniques have optimal communication complexity. Unfortunately, bijective sending places unrealistic requirements on clusters that vastly differ in size. We can address this by *partitioning* the larger-sized cluster into a set of smaller clusters, and then letting sufficient of these smaller clusters participate independent in bijective sending. This approach results in the following:

▶ **Theorem 5.** *Let $\mathfrak{S}$ be a system and let $\mathcal{C}_1, \mathcal{C}_2 \in \mathfrak{S}$. Consider the cluster-sending problem in which $\mathcal{C}_1$ sends a value v to $\mathcal{C}_2$.*

1. *If $\mathbf{n}_{\mathcal{C}} > 3\mathbf{f}_{\mathcal{C}}$, $\mathcal{C} \in \mathfrak{S}$, and $\mathfrak{S}$ has crash failures, then we can use (partitioned) bijective sending as a solution to the cluster-sending problem with optimal message complexity. These protocols solve the cluster-sending problem using $\mathcal{O}(\max(\mathbf{n}_{\mathcal{C}_1}, \mathbf{n}_{\mathcal{C}_2}))$ messages, of size $\mathcal{O}(\|v\|)$ each.*
2. *If $\mathbf{n}_{\mathcal{C}} > 4\mathbf{f}_{\mathcal{C}}$, $\mathcal{C} \in \mathfrak{S}$, and $\mathfrak{S}$ has Byzantine failures and replica sending, then we can use (partitioned) bijective sending as a solution to the cluster-sending problem with optimal replica certificate usage. These protocols solve the cluster-sending problem using $\mathcal{O}(\max(\mathbf{n}_{\mathcal{C}_1}, \mathbf{n}_{\mathcal{C}_2}))$ messages, of size $\mathcal{O}(\|v\|)$ each.*

References

1 M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Springer New York, 3th edition, 2011.

2 Maarten van Steen and Andrew S. Tanenbaum. *Distributed Systems*. Maarten van Steen, 3th edition, 2017. URL: https://www.distributed-systems.net/.

Brief Announcement: On the Correctness of Transaction Processing with External Dependency

Masoomeh Javidi Kishi
Lehigh University, Bethlehem, PA, USA
maj717@lehigh.edu

Ahmed Hassan
Alexandria University, Alexandria, Egypt[1]
ahmed.hassan@alexu.edu.eg

Roberto Palmieri
Lehigh University, Bethlehem, PA, USA
palmieri@lehigh.edu

Abstract

We briefly introduce a unified model to characterize correctness levels stronger (or equal to) serializability in the presence of application invariant. We propose to classify relations among committed transactions into data-related and application semantic-related. Our model delivers a condition that can be used to verify the safety of transactional executions in the presence of application invariant.

2012 ACM Subject Classification Theory of computation → Concurrency

Keywords and phrases Transactions, Dependency Graph, Concurrency

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.46

Funding This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-17-1-0367 and by the National Science Foundation under Grant No. CNS-1814974.

1 Introduction

When the concurrency control implementation of a transactional system is required to enforce an application-level invariant on shared data accesses (i.e., an expression that should be preserved upon every atomic update [4]), ad-hoc reasoning about its correctness is a tedious and error-prone process. Traditional (data-related) constraints (e.g., transaction conflicts) are well-formalized with established correctness levels, such as Serializability and Snapshot Isolation [1]. However, a unified model encompassing the various *external* (semantic-related) constraints that enforce application invariant has not been formalized yet.

In this brief announcement we make a step towards defining such a model. We introduce a theoretical framework that formalizes correctness levels stronger than (or equal to) serializability by defining their transaction ordering relations as a union of two sets of data and external dependency. This approach is opposed to the traditional way of defining these relations through an ad hoc analysis. This framework can be used to define an offline checker that verifies the safety of transactional executions. The intuition behind our formalization is simple. Assuming a serializable concurrency control [1], relations between transactions in an execution can be characterized as data dependency, if they are generated by data conflicts, or external dependency, if they affect the satisfaction of application invariant. This decomposition allows us to define a methodology to enrich the traditional transaction Direct

[1] Ahmed Hassan is currently affiliated with Lehigh University, USA.

© Masoomeh Javidi Kishi, Ahmed Hassan, and Roberto Palmieri;
licensed under Creative Commons License CC-BY
33rd International Symposium on Distributed Computing (DISC 2019).
Editor: Jukka Suomela; Article No. 46; pp. 46:1–46:3

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Serialization Graph (DSG) [1] with such external ordering relations. We use the formalization to introduce a safety condition that verifies correctness of transactional executions (Theorem 3).

We motivate our model by showing an example of application with associated invariant. The example mimics a simple monetary application that imposes different requirements to clients interacting from different branch locations of the bank. The application mandates the following invariant: when a transaction is issued by a client in one branch, this transaction accesses the modifications performed by the latest transactions completed on the same branch prior its starting. At the same time, the application does not require special constraints on the order of monetary transactions issued from other branches. That is, transactions from a remote branch should execute atomically and in isolation, but they might access stale data.

Suppose clients C_1 and C_2 from branch α issue two subsequent non-concurrent transactions T_1 and T_2 accessing the same bank account Ac. The first deposits $10 and the second checks the total amount of Ac and then withdraws the latest deposited amount ($10). According to the application semantics, T_2 must observe the deposit by T_1. Consider another transaction T_3, issued by a client from branch β doing auditing on accounts, including Ac. Application semantics for T_3 does not enforce any requirement on the set of transactions whose outcome should be observed, including T_1 and T_2. A serializable concurrency control would "only" guarantee a transactions order of T_1, T_2 and T_3 equivalent to some serial order. This serial order does not consider the application invariant and might order T_2 before T_1. Such a mismatch is due to the lack of application invariant representation in the concurrency control.

One solution to overcome this problem in a serializable concurrency control is to provide session guarantee [3], meaning transactions from one branch belong to the same session. This guarantee imposes an additional constraint between T_1 and T_2 where T_2 must observe the output of T_1. Clearly, T_3 would belong to a different session. The other solution would be adopting a stronger correctness level (e.g., strict serializability [1]) among all transactions, irrespective of their originating branch. An even more conservative solution is to apply external consistency [2], which brings the clients perceived order among transactions into the concurrency control so that mismatches are prevented.

With our unified model, these three correctness levels can be modeled in the same way as a combination of data-related transaction dependency, to satisfy serializability constraints, and external transaction dependency, to satisfy application invariant. This way, despite the differences among these correctness levels, our model can assess the correctness of concurrency controls that satisfy each of them by relying on a single framework.

2 Formalization

A history [1] models the interleaved execution of a set of transactions T_1, T_2, ..., T_n, as an ordered sequence of their operations (such as *read*, *write*, *abort*, *commit*). The dependency graph for a history $\mathcal{H}$, denoted as $DSG(\mathcal{H})$, represents the data-related dependency among transactions in $\mathcal{H}$. Roughly, in this graph each node is a committed transaction in $\mathcal{H}$, and each directed edge between two nodes can be of the following categories:

- *read dependency:* ($T_i \xrightarrow{\text{WR}} T_j$) A transaction T_j read-depends on T_i if a read of T_j returns a value written by T_i.
- *write dependency:* ($T_i \xrightarrow{\text{WW}} T_j$) A transaction T_j write-depends on T_i if a write of T_j overwrites a value written by T_i.
- *anti-dependency:* ($T_i \xrightarrow{\text{RW}} T_j$) A transaction T_j anti-depends on T_i if a write of T_j overwrites a value previously read by T_i.

▶ **Definition 1.** *$DSG(\mathcal{H})$ contains a set of tuples and each tuple has the following form: $(T_i, T_j, type)$. This representation shows that a directed data-related (read/write/anti-) dependency edge exists from transaction T_i to transaction T_j. $DSG(\mathcal{H}) = \{(T_i, T_j, type) : i, j \in \{1, .., n\} \wedge type \in \{RW, WW, WR\}\}$.*

Since our model focuses on correctness levels stronger than, or equal to, serializability, we recall that a history $\mathcal{H}$ is serializable if its corresponding DSG does not contain any cycle [1]. Performing an offline analysis of the DSG graph is a convenient tool for reasoning about the correctness of data-related dependencies produced by a concurrency control. However, it does not help verifying correctness of application when invariant should be preserved in addition to serializability. Our model aims at filling this gap, as follows.

▶ **Definition 2.** *An External Dependency Graph (EDG) for a given history $\mathcal{H}$, denoted as $EDG(\mathcal{H})$, determines application-level constraints. In this graph, an edge from transaction T_i to transaction T_j means an application-level requirement forces an external dependency between T_i and T_j. We say T_j externally-depends on T_i ($T_i \xrightarrow{EXT} T_j$).*

Intuitively, application invariant expressed by EDG should neither violate data-related dependency produced by the concurrency control nor include any two contradicting constraints. This observation leads to the following theorem where, informally, we consider both DSG and EDG as a single graph made by the union of them. We can check if a history is serializable and does not violate application invariant by verifying that the aforementioned single graph does not contain any cycle.

First, given a history $\mathcal{H}$ of n transactions, we define DSG, EDG, and their union as follows:

- $DSG(\mathcal{H}) = \{(V, E1) : V = \{T_i : i \in \{1, .., n\}\} \wedge E1 = \{(T_i, T_j, type) : i, j \in \{1, .., n\} \wedge type \in \{WR, WW, RW\}\}$.
- $EDG(\mathcal{H}) = \{(V, E2) : V = \{T_i : i \in \{1, .., n\}\} \wedge E2 = \{(T_i, T_j, type) : i, j \in \{1, .., n\} \wedge type \in \{EXT\}\}$.
- $DSG(\mathcal{H}) \cup EDG(\mathcal{H}) = (V, E1 \cup E2)$.

We now define our new *External Serializability* consistency level. We call a history $\mathcal{H}$ Externally Serializable (or EC-SR) if: *1)* it is serializable, and *2)* external dependency defined by the edges of its EDG are not violated. To prove that, it is necessary and sufficient to show that the union of its DSG, built from the concurrency control implementation, with its EDG, built from application invariant, does not have any cycle. We formalize that in the following theorem (the proof is intuitive and omitted due to space limitations):

▶ **Theorem 3.** *A history $\mathcal{H}$ satisfies EC-SR iff $DSG(\mathcal{H}) \cup EDG(\mathcal{H})$ does not have any cycle. A concurrency control CC satisfies EC-SR iff all the histories produced by CC are EC-SR.*

References

1. Atul Adya. Weak consistency: a generalized theory and optimistic implementations for distributed transactions, 1999.
2. James C Corbett et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.
3. Khuzaima Daudjee and Kenneth Salem. Lazy database replication with ordering guarantees. In *ICDE*, pages 424–435. IEEE, 2004.
4. Tim Harris and Simon Jones. Transactional memory with data invariants, 2006.

Brief Announcement: Towards Byzantine Broadcast in Generalized Communication and Adversarial Models

Chen-Da Liu-Zhang
Department of Computer Science, ETH Zurich, Switzerland
lichen@inf.ethz.ch

Varun Maram
Department of Computer Science, ETH Zurich, Switzerland
vmaram@inf.ethz.ch

Ueli Maurer
Department of Computer Science, ETH Zurich, Switzerland
maurer@inf.ethz.ch

Abstract

Byzantine broadcast is a primitive which allows a specific party to distribute a message consistently among n parties, even if up to t parties exhibit malicious behaviour. In the classical model with a complete network of bilateral authenticated channels, the seminal result of Pease et al. [6] shows that broadcast is achievable if and only if $t < n/3$. There are two generalizations suggested for the broadcast problem – w.r.t. the adversarial model and the communication model. Fitzi and Maurer [2] consider a (non-threshold) *general adversary* that is characterized by the subsets of parties that could be corrupted, and show that broadcast can be realized from bilateral channels if and only if the union of no three possible corrupted sets equals the entire set of n parties. On the other hand, Considine et al. [1] extend the standard model of bilateral channels with the existence of b-minicast channels that allow to locally broadcast among any subset of b parties; the authors show that in this enhanced model of communication, secure broadcast tolerating up to t corrupted parties is possible if and only if $t < \frac{b-1}{b+1}n$. These generalizations are unified in the work by Raykov [5], where a tight condition on the possible corrupted sets such that broadcast is achievable from a complete set of b-minicasts is shown.

This paper investigates the achievability of broadcast in *general networks*, i.e., networks where only some subsets of minicast channels may be available, thereby addressing open problems posed in [4, 5]. Our contributions include: 1) proposing a hierarchy over all possible general adversaries for a clean analysis of the broadcast problem in general networks, 2) showing the infeasibility of a prominent technique – used to achieve broadcast in general 3-minicast networks [7] – with regard to higher b-minicast networks, and 3) providing some necessary conditions on general networks for broadcast to be possible while tolerating general adversaries.

2012 ACM Subject Classification Theory of computation → Cryptographic protocols; Theory of computation → Distributed algorithms

Keywords and phrases broadcast, partial broadcast, minicast, general adversary, general network

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.47

1 Motivation

To the best of our knowledge, current works on the achievability of broadcast in general networks [7, 4] focus on the problem of Byzantine agreement for the concrete case of 3-minicast channels, and mainly against a threshold adversary in the range $n/3 \leq t < n/2$. We continue the line of research w.r.t. general b-minicast channels. We remark that – as noted in [1] – when $b > 3$, perfectly secure broadcast can be realized even when there is no honest majority, in contrast to Byzantine agreement.

© Chen-Da Liu-Zhang, Varun Maram, and Ueli Maurer;
licensed under Creative Commons License CC-BY

33rd International Symposium on Distributed Computing (DISC 2019).
Editor: Jukka Suomela; Article No. 47; pp. 47:1–47:3

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2 Models

Notation. Let $P = \{P_1, \ldots, P_n\}$ be a set of n parties. We say that a list $\mathcal{S} = (\mathcal{S}_0, \ldots, \mathcal{S}_{k-1})$ is a k-partition of P if $\bigcup_{i=0}^{k-1} \mathcal{S}_i = P$ and all $\mathcal{S}_i$ and $\mathcal{S}_j$ are pair-wise disjoint. In addition, we denote the set of parties from P *minus* the two sets $\mathcal{S}_i$ and $\mathcal{S}_j$ from the k-partition $\mathcal{S}$ as $\mathcal{S}_{\downarrow i,j} := P \setminus (\mathcal{S}_{i \bmod k} \cup \mathcal{S}_{j \bmod k})$.

Adversary. We assume the existence of a central adversary that corrupts a subset of parties at the onset of a protocol execution. Corrupted parties are *Byzantine*, i.e. can behave in an arbitrary way. We consider a general adversary structure $\mathcal{A}$ [3], which specifies the possible subsets of parties that the adversary can corrupt. We require that $\mathcal{A}$ be monotone, i.e., $\forall\, a,\ a'\ \ (a \in \mathcal{A})$ and $(a' \subseteq a) \implies a' \in \mathcal{A}$. In this paper, we are interested in adversary structures which satisfy the k-chain condition [5].

▶ **Definition 1.** *An adversary structure $\mathcal{A}$ is said to contain a k-chain if there exists a proper k-partition $\mathcal{S} = (S_0, \ldots, S_{k-1})$ of the party set P such that $\forall i \in [0, k-1]\ \mathcal{S}_{\downarrow i,i+1} \in \mathcal{A}$. An adversary structure is k-chain-free if it does not have a k-chain.*

Communication Network. A general network $\mathcal{N}$ among a set of parties $\mathcal{P}$ is a monotone[1] set of subsets of P. Given a general network $\mathcal{N}$, we have $\{P_{i_1}, \ldots, P_{i_k}\} \in \mathcal{N}$ if and only if there is a partial broadcast channel that allows broadcast to be realized locally among $\{P_{i_1}, \ldots, P_{i_k}\}$ – such a channel is also known as k-*minicast channel* [5]. As instantiations, the classical model with bilateral channels [6] corresponds to the network structure $\mathcal{N}$, where $\mathcal{N}$ contains all possible subsets of P with size 2; the complete b-minicast model [5] is a network structure which contains all partial broadcasts of size at most b.

3 Our Results

We extend results for general 3-minicast networks to general b-minicast networks and address open questions posed in both of the papers [4, 5], i.e. to study broadcast achievability in general communication models where only a subset of b-minicast channels may be available.

Hierarchy of Adversary Structures. We propose a hierarchy of adversary structures based on the chain terminology introduced in [5]. This allows us to analyze the feasibility of broadcast in smoothly evolving minicast models in a meaningful way. Recall that in the complete b-minicast communication model, broadcast tolerating adversary structure $\mathcal{A}$ is achievable if and only if $\mathcal{A}$ is $(b+1)$-chain-free [5]. Let the *weakest* adversary class be $\mathfrak{A}^{(0)} = \{\mathcal{A} \subseteq 2^P \mid \mathcal{A} \text{ is 3-chain-free}\}$ where broadcast is achievable with only bilateral channels, and the *strongest* adversary class be $\mathfrak{A}^{(n)} = \{\mathcal{A} \subseteq 2^P \mid \mathcal{A} \text{ contains an } n\text{-chain}\}$ where broadcast is not possible among the n parties unless we assume a global broadcast primitive in the first place. The subsequent classes of adversary structures in-between are defined as: $\forall b \in [3, n-1]$, $\mathfrak{A}^{(b)} = \{\mathcal{A} \subseteq 2^P \mid \mathcal{A} \text{ contains a } b\text{-chain and is } (b+1)\text{-chain-free}\}$. One can order the adversary classes as follows: $\mathfrak{A}^{(0)} \leq \mathfrak{A}^{(3)} \leq \mathfrak{A}^{(4)} \leq \ldots \leq \mathfrak{A}^{(n)}$. This forms a partition over all adversary structures, since for $b > 3$, any $\mathcal{A} \in \mathfrak{A}^{(b)}$ also contains an implicit $(b-1)$-chain (and lower). Observe that given an adversary $\mathcal{A} \in \mathfrak{A}^{(b)}$, broadcast tolerating $\mathcal{A}$ is impossible in the complete $(b-1)$-minicast model, but is achievable in the complete b-minicast model. This allows us to study the b-minicast channels that play an essential role in realizing broadcast against this particular adversary. We do not consider

[1] If $N \in \mathcal{N}$ and $N' \subseteq N$ then $N' \in \mathcal{N}$.

stronger classes, e.g. $\mathfrak{A}^{(b+1)}$, because [5] shows that broadcast is not achievable under these adversaries even if all b-minicast channels are available. Also, regarding weaker classes such as $\mathfrak{A}^{(b-1)}$, we know that there already exist protocols that implement secure broadcast in the complete $(b-1)$-minicast model without any b-minicast channel.

Naive Emulation of Virtual Parties. We discuss the limitations of an application of the *virtual emulation* technique to construct broadcast protocols in a (possibly incomplete) b-minicast network. The technique involves generating a new set of *virtual parties* V which are emulated by the original party set P. For example, [7] addresses the problem of achieving broadcast in a general 3-minicast network by using the available 3-minicasts to emulate virtual parties, thereby reducing the original problem to that of implementing broadcast among $P \cup V$ in the underlying communication model with bilateral channels that is secure against an *extended* adversary structure (to account for corrupted virtual parties). We show that this kind of strategy is not applicable for general b-minicast channels since we deal with significantly stronger adversaries. More concretely, if $\mathcal{A}_P$ is an adversary structure with respect to P that contains a b-chain and is $(b+1)$-chain-free (i.e., $\mathcal{A}_P \in \mathfrak{A}^{(b)}$), we prove that – irrespective of the subset of b-minicasts available for emulation in the communication model, the extended adversary $\mathcal{A}_{P\cup V}$ contains a b-chain, and thus, there does not exist any protocol among $P \cup V$ that achieves secure broadcast in the reduced $(b-1)$-minicast model.

▶ **Lemma 2.** *For $b > 3$: given an adversary structure $\mathcal{A}_P \in \mathfrak{A}^{(b)}$, for any possible set of virtual parties V emulated using b-minicast channels in the communication model, the corresponding extended adversary $\mathcal{A}_{P\cup V}$ contains a b-chain.*

Essential Partial Broadcasts. We identify some *types* of b-minicast channels that are essential for secure broadcast to be possible against general adversaries. The following characterization of partial broadcast channels also allows us to derive a lower bound on the number of b-minicasts required for the parties to broadcast globally in any general network.

▶ **Theorem 3.** *Secure broadcast on a general network $\mathcal{N}$ tolerating any general adversary $\mathcal{A} \in \mathfrak{A}^{(b)}$ is possible only if: for every b-chain in $\mathcal{A}$, namely $\mathcal{P} = (\mathcal{P}_0, \ldots, \mathcal{P}_{b-1})$, there is a b-minicast channel in $\mathcal{N}$ that has non-empty intersection with the sets $\mathcal{P}_0, \ldots, \mathcal{P}_{b-1}$.*

References

1. J. Considine, M. Fitzi, M. Franklin, L. A. Levin, U. Maurer, and D. Metcalf. Byzantine agreement given partial broadcast. *Journal of Cryptology*, 18(3):191–217, July 2005.
2. M. Fitzi and U. M. Maurer. Efficient Byzantine agreement secure against general adversaries. In S. Kutten, editor, *DISC*, volume 1499 of *LNCS*, pages 134–148. Springer, 1998.
3. Martin Hirt and Ueli Maurer. Complete characterization of adversaries tolerable in secure multi-party computation. In *PODC*, volume 97, pages 25–34, 1997.
4. A. Jaffe, T. Moscibroda, and S. Sen. On the price of equivocation in Byzantine agreement. In D. Kowalski and A. Panconesi, editors, *ACM PODC '12*, pages 309–318. ACM, 2012.
5. Raykov P. Broadcast from minicast secure against general adversaries. In M. M. Halldórsson, K. Iwama, N. Kobayashi, and B. Speckmann, editors, *ICALP 2015*, volume 9135 of *LNCS*, pages 701–712. Springer, Berlin, Germany, 2015.
6. M. C. Pease, R. E. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.
7. D. V. S. Ravikant, M. Venkitasubramaniam, V. Srikanth, K. Srinathan, and C. P. Rangan. On byzantine agreement over (2,3)-uniform hypergraphs. In R. Guerraoui, editor, *DISC 2004*, volume 3274 of *LNCS*, pages 450–464. Springer, 2004.

Brief Announcement: Integrating Temporal Information to Spatial Information in a Neural Circuit

Nancy Lynch
Massachusetts Institute of Technology, Cambridge, MA, United States
lynch@csail.mit.edu

Mien Brabeeba Wang
Massachusetts Institute of Technology, Cambridge, MA, United States
brabeeba@mit.edu

Abstract

In this paper, we consider networks of deterministic spiking neurons, firing synchronously at discrete times. We consider the problem of translating temporal information into spatial information in such networks, an important task that is carried out by actual brains. Specifically, we define two problems: "First Consecutive Spikes Counting" and "Total Spikes Counting", which model temporal-coding and rate-coding aspects of temporal-to-spatial translation respectively. Assuming an upper bound of T on the length of the temporal input signal, we design two networks that solve two problems, each using $O(\log T)$ neurons and terminating in time $T + 1$. We also prove that these bounds are tight.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Theory of computation → Distributed algorithms; Applied computing → Biological networks

Keywords and phrases Spiking Neural Network, Distributed Algorithm, Biological Networks

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.48

Related Version A full version of the paper is available at https://arxiv.org/abs/1903.01217.

Funding This material is based upon work supported by the National Science Foundation under grant no. CCF-1810758, CCF-1461559, and CCF-0939370.

Introduction. Algorithms in the brain are inherently distributed. Although each neuron has relatively simple dynamics, as a distributed system, a network of neurons exhibits strong computational power. Recently, there has been increasing interest in using biologically plausible spiking neuronal dynamics to solve different computational problems [4, 1]. In this paper, we consider a network of spiking neurons with a deterministic synchronous firing rule operating in discrete time, similar to that of Lynch, et al. [4], in order to simplify the analysis and focus on the computational principles.

One of the most important questions in neuroscience is how humans integrate information over time. Sensory inputs such as visual and auditory stimulus are inherently temporal; however, brains are able to integrate the temporal information to a single concept, such as a moving object in a visual scene, or a word in a sentence. There are two kinds of neuronal codings: rate coding and temporal coding. Rate coding is a neural coding scheme assuming that most of the information is coded in the firing rate of the neurons. It is most commonly seen in muscle in which the higher firing rates of motor neurons correspond to higher intensity in muscle contraction. On the other hand, rate coding cannot be the only neural coding brains employ. A fly is known to react to new stimuli and change its direction of flight within 30-40 ms. There is simply not enough time for neurons to compute the firing rate which is the average of spike counts over an interval. Therefore, neuroscientists have proposed the idea of temporal coding, assuming the information is coded in the temporal firing patterns. One of the popular temporal codings is the "first-to-spike" coding. It has been shown that the timing of the first spike encodes most information of an image in retinal cells [2].

© Nancy Lynch and Mien Brabeeba Wang;
licensed under Creative Commons License CC-BY
33rd International Symposium on Distributed Computing (DISC 2019).
Editor: Jukka Suomela; Article No. 48; pp. 48:1–48:3

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

We propose two toy problems to model how brains extract information from different codings. "First Consecutive Spikes Counting" (FCSC) counts the first consecutive interval of spikes, which is equivalent to counting the distance between the first two spikes, a prevalent temporal coding scheme in sensory cortex. "Total Spikes Counting" (TSC) counts the number of the spikes over an arbitrary interval, which is an example of rate coding.

In this paper, we design two networks that solve the above two problems in time $T+1$ with $O(\log T)$ neurons. We also show that our time bounds are tight. We remark that although our problems are biologically inspired, the optimal solutions we propose are not biologically plausible. Our networks are not noise tolerant, whereas the actual neuronal dynamics are highly noisy. However, we hope that our results can demonstrate an important computational principle: unstable intermediate states can carry temporal information and then converge to a stable representation efficiently.

In other work on spiking neural networks, Hitron and Parter [3] tackled a similar problem to our TSC problem. Our results differ in three ways. First, our network has time bound $T+1$ while theirs is $T+O(\log T)$. Second, we provide a time lower bound result and show that our time bound is optimal. Third, they additionally consider an approximate version of the problem while we focus on the exact version of the problem.

Model. We consider a Spiking Neural Network (SNN) model with deterministic synchronous firing at discrete times. Our network structure consists of a directed graph (V,E) with bias, $b: V \to \mathbb{R}$ and weight function, $w: E \to \mathbb{R}$. In this paper, we fix input neuron $x \in V$ and m output neurons $\{y_i\}_{0 \le i < m} \subset V$. The dynamics of neuron $z \in V$ at each time step $t \ge 1$ is governed by

$$z^{(t)} = \Theta(\sum_{y \in P_z} w_{yz} y^{(t-1)} - b_z).$$

where $z^{(t)}$ is the indicator function of neuron z firing at time t. P_z is the set of presynaptic neurons of z, and Θ is a nonlinearity. Here we take Θ as the Heaviside function given by $\Theta(x) = 1$ if $x > 0$ and 0 otherwise. At $t = 0$, we let $z^{(0)} = 0$ if z is not an input neurons.

The First Consecutive Spikes Counting(T) (FCSC(T)). Given an input neuron x and the max input length T, we consider any input firing sequence such that for all $t \ge T$, $x^{(t)} = 0$. Define L_x to be the length of the first consecutive spikes interval. Then we say a network of neurons solves FCSC(T) in time t' with m' neurons if there exists an injective function $F: \{0, \cdots, T\} \to \{0,1\}^m$ which serves as an encoding of the count such that for all x and for all $t, t \ge t'$ we have $y^{(t)} = F(L_x)$ and the network has m' total neurons.

Intuitively, FCSC serves as a toy model for encoding distance between spikes, a prevalent temporal coding in sensory cortex. For mathematical convenience, we model the problem as counting the distance between non-spikes which is mathematically equivalent to counting the distance between spikes in our model.

The Total Spikes Counting(T) (TSC(T)). Given an input neuron x and the max input length T, we consider any input firing sequence such that for all $t \ge T$, $x^{(t)} = 0$. Define L_x to be the total number of spikes in the sequence. Then we say a network of neurons solves TSC(T) in time t' with m' neurons if there exists an injective function $F: \{0, \cdots, n\} \to \{0,1\}^m$ which serves as an encoding of the count such that for all x and for all $t, t \ge t'$ we have $y^{(t)} = F(L_x)$ and the network has m' total neurons.

Intuitively, TSC serves as a toy model for rate coding implemented by spiking neural networks, because the network is able to extract the rate information by counting the number of spikes over arbitrary intervals.

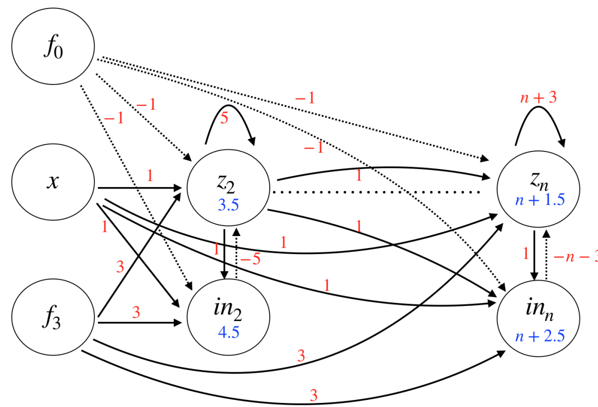


Figure 1 Modified Binary Counter Part of the TSC Network.

Results. Our contributions in the paper are to design networks that solve these two problems respectively with matching lower bounds in terms of the number of neurons.

▶ **Theorem 1.** *There exist networks with $O(logT)$ neurons that solve the FCSC(T) and TSC(T) problems, both in time $T + 1$.*

Our FCSC network does binary counting by precomputing simultaneous carrying and then captures the counter into persistent neurons. TSC network has further complications because it requires the network to count spikes regardless of intervening non-spikes. In particular, TSC presents an interesting difficulty: there are conflicting objectives between maintaining the count when no spike arrives and updating the count when a spike arrives. To overcome this difficulty, we allow the network to enter an unstable intermediate state which carries the information of the count. The intermediate state then converges to a stable state that represents the count after a computation step without inputs. To be concrete, our TSC network contains a modified binary counter (Figure 1) which does delayed simultaneous carrying but we need to handle the subtle behaviors of delay carefully in our design. Our time lower bound result shows that this delay is indeed necessary.

▶ **Theorem 2.** *There is no network with $o(T)$ neurons that solves FCSC(t) problem in time t for all $0 \leq t \leq T$. The same holds for the TSC problem.*

Intuitively, the proof of the time lower bound uses the fact that if the network has to solve the problem without delay, then it must stabilize immediately at each time step. Therefore, the neurons that fire at the last round will continue firing. By injectivity of the representation, we can conclude that the network can at most count up to the network size.

References

1 Chi-Ning Chou, Kai-Min Chung, and Chi-Jen Lu. On the Algorithmic Power of Spiking Neural Networks. *ITCS*, 2019.

2 T. Gollisch and M. Meister. Rapid neural coding in the retina with relative spike latencies. *Science*, 319:1108–1111, 2008.

3 Yael Hitron and Merav Parter. Counting to Ten with Two Fingers: Compressed Counting with Spiking Neurons. *arXiv:1902.10369 [cs.NE]*, 2019. `arXiv:1902.10369`.

4 Nancy A. Lynch, Cameron Musco, and Merav Parter. Computational Tradeoffs in Biological Neural Networks: Self-Stabilizing Winner-Take-All Networks. *ITCS*, 2017.

Brief Announcement: Faster Asynchronous MST and Low Diameter Tree Construction with Sublinear Communication

Ali Mashreghi
Department of Computer Science, University of Victoria, BC, Canada
ali.mashreghi87@gmail.com

Valerie King
Department of Computer Science, University of Victoria, BC, Canada
val@uvic.ca

Abstract

Building a spanning tree, minimum spanning tree (MST), and BFS tree in a distributed network are fundamental problems which are still not fully understood in terms of time and communication cost. The first work to succeed in computing a spanning tree with communication sublinear in the number of edges in an asynchronous CONGEST network appeared in DISC 2018. That algorithm which constructs an MST is sequential in the worst case; its running time is proportional to the total number of messages sent. Our paper matches its message complexity but brings the running time down to linear in n. Our techniques can also be used to provide an asynchronous algorithm with sublinear communication to construct a tree in which the distance from a source to each node is within an additive term of $\sqrt{n}$ of its actual distance.

2012 ACM Subject Classification Theory of computation → Distributed computing models

Keywords and phrases Distributed Computing, Minimum Spanning Tree, Broadcast Tree

Digital Object Identifier 10.4230/LIPIcs.DISC.2019.49

Related Version https://arxiv.org/abs/1907.12152

Funding *Ali Mashreghi*: Funded with an NSERC grant.
Valerie King: Funded with an NSERC grant.

1 Problem and Results

A distributed network of processes can be represented as an undirected graph $G = (V, E)$, where $|V| = n$ and $|E| = m$. Each node corresponds to a process and each edge corresponds to a communication link between the two processes. The nodes can communicate only by passing messages to each other. Computing the spanning tree and the minimum spanning tree (MST) are problems of fundamental importance in distributed computing. Efficient solutions for building these trees directly improve the solution to other distributed computing problems or at least provide valuable insights. Leader election, counting, and shortest path tree are examples of such problems. The breadth-first search tree (BFS) is also important; it can be used to simulate a synchronous algorithm in an asynchronous network.

The problem of constructing an MST in a distributed network has been studied for many years. In the earlier works, researchers focused on improving the time complexity since it was believed that any spanning tree algorithm in the CONGEST model would require $\Omega(m)$ messages (See [2]). After the algorithm of King et al. [5] which constructs the MST in the synchronous CONGEST model in $\tilde{O}(n)$ time and messages, there has been renewed interest in message complexity. Mashreghi and King [6] achieved the first algorithm to compute a spanning tree in an asynchronous CONGEST model with $o(m)$ communication complexity

© Ali Mashreghi and Valerie King;
licensed under Creative Commons License CC-BY
33rd International Symposium on Distributed Computing (DISC 2019).
Editor: Jukka Suomela; Article No. 49; pp. 49:1–49:3

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

when m is sufficiently large. However, the time complexity of their algorithm matches its communication complexity, $\tilde{O}(n^{3/2})$; in the worst case, their algorithm essentially operates in a sequential manner.

In this work, we match the message complexity of [6] but bring the running time down to $O(n)$, a time which matches the time of the fastest known asynchronous MST algorithms that use $\Theta(m)$ communication [1]. Full version of our paper is available on Arxiv [7].

The classic Layered BFS algorithm for the asynchronous CONGEST network uses $O(D^2)$ time and $O(Dn+m)$ messages, where D is the diameter of the network. We show how to construct a "nearly BFS" tree in this model with sublinear number of messages (for small enough D, and sufficiently large m) such that for each node, the distance from the source node is within an additive term of $O(\sqrt{n})$ from its actual distance in the network. Such a tree can be used to simulate a synchronous algorithm in an asynchronous network with an overhead of $O(D+\sqrt{n})$ time per step. To the best of our knowledge, there is no previously known algorithm to construct a low diameter tree using a sublinear number of messages in an asynchronous network. Specifically, we show:

▶ **Theorem 1.** *There exists an asynchronous algorithm in the KT1 CONGEST model that, w.h.p. computes the MST in $O(n)$ time and with $O(\min\{m, n^{3/2}\log^2 n\})$ messages.*

This result achieves communication sublinear in m when m is sufficiently large, and is optimal for time when the diameter is $\Theta(n)$. We also prove the following more general theorem.

▶ **Theorem 2.** *Given an asynchronous MST algorithm with time $T(n,m)$ and message complexity of $M(n,m)$ in the KT1 CONGEST model, w.h.p., the MST in an asynchronous network can be constructed in $O(n^{1-2\epsilon}+T(n,n^{3/2+\epsilon}))$ time and $\tilde{O}(n^{3/2+\epsilon}+M(n,n^{3/2+\epsilon}))$ messages, for $\epsilon \in [0,1/4]$.*

For the BFS problem we show:

▶ **Theorem 3.** *In an asynchronous KT1 CONGEST model, a network with diameter D can construct a spanning tree with $O(D+\sqrt{n})$ diameter using time $O(D^2+n)$ and messages $\tilde{O}(n^{3/2}+nD)$.*

Model

We consider the asynchronous CONGEST model. Messages sent by nodes are delayed arbitrarily. Communication is event-driven, in that actions are taken upon receiving a message or waking up. We assume that all nodes wake up at the start. Time complexity in the asynchronous communication model is the worst-case execution time, if each message takes at most one time unit to traverse one edge. The time for computations within a node is not considered.

All nodes have knowledge of n, the size of the network, within a constant factor. In the CONGEST model, each message has $O(\log n)$ bits. ID's are unique and are taken from the range of $[1, poly(n)]$. We assume that messages to a receiver are numbered by a sender in the order in which they are sent to it, so that a node receiving the message can wait for the message from a sender with the sender's next number before acting. W.l.o.g., we assume that the edge weights are unique and therefore, the MST is unique.

Nodes initially know their own ID and the ID of their neighbors. This is known as the KT1 model and is considered by some to be the standard model of distributed computing [8]. In weighted graphs, initially, nodes only know the weight of their incident edges in the input graph G. We assume that all nodes wake up at the same time. For constructing a subgraph like MST, the objective is that, upon termination of the protocol, all nodes must know which of their incident edges belong to the subgraph.

2 Outline of algorithms

We achieve some parallelism by starting with *initial* fragments of height 1, formed by high degree nodes and star nodes in parallel. A node is *high-degree* if its degree is at least $\sqrt{n}\log^2 n$. Otherwise, it is a *low-degree* node. A node selects itself to be a *star node* with probability of $\frac{c}{\sqrt{n}\log n}$ where c is a constant dependent on c' so that each high-degree node is adjacent to a star node with probability $1 - 1/n^{c'}$. Let G' be the subgraph on G induced by all high-degree and star nodes. We can construct a spanning forest F on G', from the initial fragments by adding $O(\sqrt{n}/\log n)$ edges such that the sum of the diameter of all trees in this forest is $O(\sqrt{n}/\log n)$. Thus we can afford to add these edges sequentially in the worst case, spending a time per edge proportional to $\log^2 n *$ (diameter of a maximum tree), for a total of $O(n)$ time. To find minimum outgoing edges, one approach is to have nodes test all of their incident edges, in the order of weight, to see whether an edge is outgoing. This results in $\Omega(m)$ messages. King et al. [5] provided an *asynchronous* subroutine, called FindAny, that with constant probability, finds an edge leaving a fragment T and uses only $\tilde{O}(|T|)$ messages, where $|T|$ is the number of nodes in T. The algorithm has three parts as follows:

1. Initial fragments are formed in parallel consisting of star nodes and their high degree neighbors. Our new algorithm MAXIMALTREE incorporates the asynchronous waiting technique of [6] to find the edges in F and enable each high-degree node to learn its low-degree neighbors while building a spanning forest F on G' from these initial fragments. To construct the nearly BFS tree, the nodes run Gallager's Layered BFS algorithm [3] on G_{sparse}, the subgraph of edges in F and those incident to at least one low-degree node.
2. We compute the minimum weight spanning forest F_{min} on G'. We use an idea of [4], along with the fact that the obtained trees in part (1) all have diameter of $O(\sqrt{n})$. This is done using the low-diameter trees in F, to simulate the MST algorithm of [4] in the asynchronous model on each connected component of G'.
3. We define S_{min} to be the edges in F_{min} and the edges incident to at least one low-degree node. We run an asynchronous MST algorithm with $O(n)$ time and $O(m)$ message complexity (e.g. [1]) on S_{min}. The result is the MST of G.

References

1 Baruch Awerbuch. Optimal distributed algorithms for minimum weight spanning tree, counting, leader election, and related problems. In *STOC*, pages 230–240, 1987.

2 Baruch Awerbuch, Oded Goldreich, Ronen Vainish, and David Peleg. A trade-off between information and communication in broadcast protocols. *Journal of the ACM*, 37(2):238–256, 1990.

3 R.G. Gallager. Distributed minimum hop algorithms. Technical Report LIDS-P-1175, MIT Laboratory for Information and Decision Systems, 1982.

4 Mohsen Ghaffari and Fabian Kuhn. Distributed MST and Broadcast with Fewer Messages, and Faster Gossiping. In *DISC*, pages 30:1–30:12, 2018.

5 Valerie King, Shay Kutten, and Mikkel Thorup. Construction and impromptu repair of an MST in a distributed network with o(m) communication. In *PODC*, pages 71–80, 2015.

6 Ali Mashreghi and Valerie King. Broadcast and Minimum Spanning Tree with $o(m)$ Messages in the Asynchronous CONGEST Model. In *DISC*, volume 121, pages 37:1–37:17, 2018.

7 Ali Mashreghi and Valerie King. Faster asynchronous MST and low diameter tree construction with sublinear communication. *arXiv e-prints*, July 2019. `arXiv:1907.12152`.

8 David Peleg. Distributed computing. *SIAM Monographs on discrete mathematics and applications*, 5, 2000.