Efficient Circuit Simulation in MapReduce

Fabian Frei
Department of Computer Science, ETH Zürich, Universitätstrasse 6, CH-8006 Zürich, Switzerland
fabian.frei@inf.ethz.ch

Koichi Wada
Department of Applied Informatics, Hosei University, 3-7-2 Kajino, 184-8584 Tokyo, Japan
wada@hosei.ac.jp

Abstract

The MapReduce framework has firmly established itself as one of the most widely used parallel computing platforms for processing big data on tera- and peta-byte scale. Approaching it from a theoretical standpoint has proved to be notoriously difficult, however. In continuation of Goodrich et al.'s early efforts, explicitly espousing the goal of putting the MapReduce framework on footing equal to that of long-established models such as the PRAM, we investigate the obvious complexity question of how the computational power of MapReduce algorithms compares to that of combinational Boolean circuits commonly used for parallel computations. Relying on the standard MapReduce model introduced by Karloff et al. a decade ago, we develop an intricate simulation technique to show that any problem in $\mathcal{NC}$ (i.e., a problem solved by a logspace-uniform family of Boolean circuits of polynomial size and a depth polylogarithmic in the input size) can be solved by a MapReduce computation in $O(T(n)/\log n)$ rounds, where n is the input size and $T(n)$ is the depth of the witnessing circuit family. Thus, we are able to closely relate the standard, uniform $\mathcal{NC}$ hierarchy modeling parallel computations to the deterministic MapReduce hierarchy $\mathcal{DMRC}$ by proving that $\mathcal{NC}^{i+1} \subseteq \mathcal{DMRC}^i$ for all $i \in \mathbb{N}$. Besides the theoretical significance, this result has important applied aspects as well. In particular, we show for all problems in $\mathcal{NC}^1$ – many practically relevant ones, such as integer multiplication and division and the parity function, being among these – how to solve them in a constant number of deterministic MapReduce rounds.

2012 ACM Subject Classification Theory of computation → Complexity classes; Computing methodologies → MapReduce algorithms; Theory of computation → Circuit complexity; Theory of computation → MapReduce algorithms; Software and its engineering → Ultra-large-scale systems

Keywords and phrases MapReduce, Circuit Complexity, Parallel Algorithms, Nick's Class $\mathcal{NC}$

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2019.52

Related Version The full version of this paper including all figures and proofs is freely available at `http://arxiv.org/abs/1907.01624`.

Funding *Koichi Wada*: Research done in part during a supported visit at ETH Zürich and partly supported by JSPS KAKENHI No. 17K00019 and by the Japan Science and Technology Agency (JST) SICORP (Grant#JPMJSC1806).

Acknowledgements We thank the anonymous reviewers for their helpful comments.

1 Introduction

Despite the overwhelming success of the MapReduce framework in the big data industry and the great attention it has garnered ever since its inception over a decade ago, theoretical results about it have remained scarce in the literature. In particular, it is very natural to ask how powerful exactly MapReduce computations are in comparison to the traditional models of parallel computations based on circuits; a question that has practical implications as well. The answers have proved to be very elusive, however. In this paper, we show how MapReduce programs can efficiently simulate circuits used for parallel computations, thus tying these two worlds together more tightly.

30th International Symposium on Algorithms and Computation (ISAAC 2019).
Editors: Pinyan Lu and Guochuan Zhang; Article No. 52; pp. 52:1–52:21
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In this section we first provide an introduction to the concept of MapReduce, then present the related work, and finally describe our contribution. In Section 2, we will formally define the traditional models of parallel computing and the MapReduce model. In Section 3, we then derive our main results. Section 4 concludes the paper with a short summary and a discussion of our findings, outlining opportunities for future research.

1.1 Background and Motivation

In recent years the amount of data available and demanding analysis has experienced an astonishing growth. The amount of memory in commercially available servers has also grown at a remarkable pace over the past decade; it is now exceeding tera- and even peta-bytes. Despite the considerable advances in the availability of computational power, traditional approaches remain insufficient to cope with such huge amounts of data. A new form of parallel computing has become necessary to deal with these enormous quantities of available data. The MapReduce framework has been attracting great interest due to its suitability for processing massive data-sets. This framework was originally developed by Google [5], but an open source implementation called Hadoop has recently been developed and is currently used by over a hundred companies, including Yahoo!, Facebook, Adobe, and IBM [19].

MapReduce differs substantially from previous models of parallel computation in that it combines aspects of both parallel and sequential computation. Informally, a MapReduce computation can be described as follows.

The input is a multiset of *key-value pairs* $\langle k; v\rangle$. In a first step, the *map step*, each of these key-value pairs is separately and independently transformed into an entire multiset of key-value pairs by a *map function* μ. In the next step, the *shuffle step*, we collect all key-value pairs from the multisets that have been produced in the previous step, group them by their keys, and collapse each group $\{\langle k; v_1\rangle, \langle k; v_2\rangle, \ldots\}$ of pairs containing the same key into a single key-value pair $\langle k; \{v_1, v_2, \ldots\}\rangle$ consisting of said key and a list of the associated values. In a third step, the *reduce step*, a *reduce function* ρ transforms the list of values in each key-value pair $\langle k; \{v_1, v_2, \ldots\}\rangle$ into a new list $\{v'_1, v'_2, \ldots\}$. Again, this is done separately and independently for each pair. The final output consists of the pairs $\{\langle k; v'_1\rangle, \langle k; v'_2\rangle, \ldots\}$ for each key k. The different instances that implement the reduce function for the different groups of pairs are called reducers. Analogously, mappers are instances of the map function.

The three steps described above constitute one *round* of the MapReduce computation and transform the input multiset into a new multiset of key-value pairs. A complete MapReduce computation consists of any given number of rounds and acts just as the composition of the single rounds. The shuffle step works the same way every time; the map and reduce functions, however, may change from round to round. A MapReduce computation with R rounds is therefore completely described by a list $\mu_1, \rho_1, \mu_2, \rho_2, \ldots, \mu_R, \rho_R$ of map and reduce functions. In both the map step and the reduce step, the input pairs can be processed in parallel since the map and reduce functions act independently on the pairs and groups of pairs, respectively. These steps therefore capture the parallel aspect of a MapReduce computation, whereas the shuffle step enforces a partial sequentiality since the shuffled pairs can be output only once the previous map step is completed in its entirety.

The MapReduce paradigm has been introduced in [5] in the context of algorithm design and analysis. A treatment as a formal computational model, however, was missing in the beginning. Later on, a number of models have emerged to deal more rigorously with algorithmic issues [7, 10, 12, 14, 15]. In this paper, our interest lies in studying the MapReduce framework from a standpoint of parallel algorithmic power by comparing it to standard models of parallel computation such as Boolean circuits and parallel random access machines

(PRAMs). A PRAM can be classified by how far simultaneous access by processors to its memory is restricted; it can be CRCW, EREW, CREW, or ERCW, where R, W, C, and E stand for Read, Write, Concurrent, and Exclusive, respectively [4]. If concurrent writing is allowed, we need to further specify how parallel writes by multiple processors to a single memory cell are handled. The most natural choice is arguably that every memory cell contains after each time step the total of all numbers assigned to it by different processors during that step. In fact, all constructions in this paper work with this treatment of simultaneous writes; we thus generally assume this model. If the context warrants it, we speak of a Sum-CRCW to make this assumption explicit.

1.2 Related Work

We briefly present and discuss the following known results on the comparative power of the MapReduce framework and PRAM models.

1. A T-time EREW-PRAM algorithm can be simulated by an $O(T)$-round MapReduce algorithm, where each reducer uses memory of constant size and an aggregate memory proportional to the amount of shared memory required by the PRAM algorithm [10, 12].
2. A P-processor, M-memory, T-time EREW-PRAM algorithm can be simulated by an $O(T)$-round, $(P+M)$-key MUD algorithm with a communication complexity of $O(\log(P+M))$ bits per key, where a MUD (massive, unordered, distributed) algorithm is a data-streaming MapReduce algorithm in the following sense: The reducers do not receive the entire list of values associated with a given key at once, but rather as a stream to be processed in one pass, using only a small working memory determining the communication complexity [7].
3. When using MapReduce computations to simulate a CRCW-PRAM instead, again with P processors and M memory, we incur an $O(\log_m(P+M))$ slowdown compared to the simulations above, where m is an upper bound on each reducer's input and output [10].

These results imply that any problem solved by a PRAM with a polynomial number of processors and in polylogarithmic time T can be simulated by a MapReduce computation with an amount of memory equal to the number of PRAM processors, and in a number of rounds equal to the computation time of even the powerful CRCW-PRAM. Since the class of problems solved by CRCW-PRAMs in time $T \in O(\log^i n)$ is equal to the class of problems solved by families of polynomial-sized combinational circuits consisting of gates with unbounded fan-in and fan-out and depth $T \in O(\log^i n)$ (often denoted $\mathcal{AC}^i$) [1], these circuits can be simulated in a MapReduce computation with a number of rounds equal to the time required by these circuits.

Since the publication of the seminal paper by Karloff et al. [12], extensive effort has been spent on developing efficient algorithms in MapReduce-like frameworks [3, 6, 13, 11, 17]. Only few relationships between the theoretical MapReduce model by [12] and classical complexity classes have been established, however; for example, any problem in $\mathcal{SPACE}(o(\log n))$ can be solved by a MapReduce computation with a constant number of rounds [8].

Recently, Roughgarden et al. [16, Theorem 6.1] described a short and simple way of simulating $\mathcal{NC}^1$ circuits with a certain class of models of parallel computation. The constraints of these models, namely the number of machines and the memory restrictions, are exactly tailored to allow for this general simulation method, however. In particular, it crucially relies on the fact that all models of this class are more powerful than the MapReduce model in that they all grant us a number of machines that is polynomial in the input size; this makes it possible to just dedicate one machine to each of the circuit gates. Such a simple simulation is impossible with MapReduce computations since the standard model due to Karloff only allows for a sublinear number of machines with sublinear memory.

1.3 Contribution

We prove that $\mathcal{NC}^{i+1} \subseteq \mathcal{DMRC}^i$ for all $i \in \{0, 1, 2, \dots\}$, where $\mathcal{DMRC}^i$ is the set of problems solvable by a deterministic MapReduce computation in $O(\log^i n)$ rounds. In the case of $\mathcal{NC}^1 \subseteq \mathcal{DMRC}^0$, which already opens up a plethora of applications on its own, the result holds for every possible choice of ε, that is, for $0 < \varepsilon \leq 1/2$. The higher levels of the hierarchy require an entirely different proof method, which yields the result for $0 < \varepsilon < 1/2$.

This is a substantial improvement over the previous results that only imply, as outlined above, the far weaker claim $\mathcal{AC}^i \subseteq \mathcal{MRC}^i$. The case $i = 1$ is of particular practical interest since $\mathcal{NC}^1 \setminus \mathcal{AC}^0$ contains plenty of relevant problems such as integer multiplication and division, the parity function, and the recognition of Dyck languages; see [1]. Our results show how to solve all of these problems with a deterministic MapReduce program in a constant number of rounds.

2 Preliminaries

We denote by $\mathbb{N} = \{0, 1, 2, \dots\}$ the natural numbers including zero and let $\mathbb{N}_+ = \mathbb{N} \setminus \{0\}$. Moreover, we let $[i] = \{0, 1, \dots, i-1\}$ denote the i first natural numbers for any $i \in \mathbb{N}_+$.

2.1 Models of Parallel Computation

In this section, we define the common complexity classes capturing the power of parallel computation; most prominently the $\mathcal{NC}$ hierarchy.

A finite set $\mathcal{B} = \{f_0, \dots, f_{|\mathcal{B}|-1}\}$ of Boolean functions $f_i : \{0,1\}^{n_i} \to \{0,1\}$ with $n_i \in \mathbb{N}$ for every $i \in [|\mathcal{B}|]$ is called a *basis*. For every $n, m \in \mathbb{N}_+$, a *(Boolean) circuit* C over the basis $\mathcal{B}$ with n inputs and m outputs is a directed acyclic graph that contains n *sources* (nodes with no incoming edges), called the *input nodes*, and m *sinks* (nodes with no outgoing edges). The *fan-in* of a node is the number of incoming edges, the *fan-out* is the number of outgoing edges. Nodes that are neither sources nor sinks are called *gates*. Each gate is labeled with a function $f_i \in \mathcal{B}$ and has fan-in n_i. It computes f_i on the input given by the incoming edges and outputs the result (either 0 or 1) to each of the outgoing edges. A basis $\mathcal{B}$ is said to be *complete* if for every Boolean function f, we can construct over the basis $\mathcal{B}$ a circuit of the described form that computes f. In the following, we use the complete basis $\mathcal{B} = \{\vee, \wedge, \neg\}$.

The *size* of a circuit C, denoted by $\mathrm{size}(C)$, is the total number of edges it contains. The *level* of a node v in a circuit C, denoted $\mathrm{level}(v)$, is defined recursively: The level of a sink is 0, and the level of a node v with nonzero fan-out is one greater than the maximum of the levels of the outgoing neighbors of v. The *depth* of C, denoted $\mathrm{depth}(C)$, is the maximum level across all nodes in C. A function $f : \{0,1\}^* \to \{0,1\}^*$ is *implicitly logspace computable* if the two mappings $(x, i) \mapsto \chi_{i \leq |f(x)|}$, where χ denotes the characteristic function, and $(x, i) \mapsto (f(x))_i$ are computable using logarithmic space. A circuit family $\{C_n\}_{n=0}^{\infty}$ is *logspace-uniform* if there is an implicitly logspace computable function mapping 1^n to the description of the circuit C_n. It is known that the class of languages that have logspace-uniform circuits of polynomial size equals $\mathcal{P}$ [1, Thm. 6.15].

For any $i \in \mathbb{N}$, the complexity class $\mathcal{NC}^i$ contains a language L exactly if there is a constant c and a logspace-uniform family of circuits $\{C_n\}_{n=0}^{\infty}$ recognizing L such that C_n has size $O(n^c)$, depth $O(\log^i n)$, and all nodes have fan-in at most 2. The union is Nick's class $\mathcal{NC} = \bigcup_{i=0}^{\infty} \mathcal{NC}^i$. We mention that there is an analogous definition of classes Nonuniform-$\mathcal{NC}^i$ that do not require logspace uniformity from the circuits; they constitute a different hierarchy.

The complexity classes $\mathcal{AC}^i$ and $\mathcal{AC} = \bigcup_{i=0}^{\infty} \mathcal{AC}^i$ are defined exactly as $\mathcal{NC}^i$ and $\mathcal{NC}$, except that the restriction of the maximal fan-in to at most 2 is omitted. Nevertheless, the restriction on the circuit size imply that the fan-in of a node is bounded by a polynomial in n. The OR gates and AND gates in such a circuit can therefore be replaced by trees of gates of fan-in at most 2 with a depth in $O(\log n)$. It follows that $\mathcal{AC}^i \subseteq \mathcal{NC}^{i+1}$ for all $i \in \mathbb{N}$ and thus $\mathcal{NC} = \mathcal{AC}$. (Analogously, we see why Nick's class can also be defined, as it often is, by upper-bounding the fan-in by an arbitrary constant greater than 2.) The inclusion $\mathcal{NC}^i \subseteq \mathcal{AC}^i$ for every $i \in \mathbb{N}$ is immediate from the definition. The first two inclusions of the resulting chain are known to be strict – namely, we have $\mathcal{NC}^0 \subsetneq \mathcal{AC}^0 \subsetneq \mathcal{NC}^1$; see [1].

Finally, we summarize the known results on how the classes of languages recognized by different PRAMs fit into the two hierarchies of $\mathcal{NC}$ and $\mathcal{AC}$. Let $\mathcal{EREW}^i$, $\mathcal{CREW}^i$ and $\mathcal{CRCW}^i$ denote the sets of problems of size n computed by EREW-PRAMs, CREW-PRAMs, and CRCW-PRAMs, respectively, with a polynomial number of processors in $O(\log^i n)$ time. For every $i \in \mathbb{N}$, we have $\mathcal{NC}^i \subseteq \mathcal{EREW}^i \subseteq \mathcal{CREW}^i \subseteq \mathcal{CRCW}^i = \mathcal{AC}^i \subseteq \mathcal{NC}^{i+1}$; see [1].

2.2 The MapReduce Model

In this section we describe the standard MapReduce model as proposed by [12]. It defines the notions of *map functions* and *reduce functions*, which are summarized under the term *primitives*. Roughly speaking, a MapReduce computing system executes primitives, interleaved with so-called *shuffle* operations. The basic data unit in these computations is an ordered pair $\langle key; value\rangle$, called *key-value pair*. In general, keys and values are just binary strings, allowing us to encode all the usual entities.

A map function is a (possibly randomized) function that takes as input a single key-value pair and outputs a finite multiset of new key-value pairs. A reduce function (again, possibly randomized) takes instead an entire set of key-value pairs $\{\langle k; v_{k,1}\rangle, \langle k; v_{k,2}\rangle, \ldots\}$, where all the keys are identical, and outputs a single key-value pair $\langle k; v'\rangle$ with that same key.

A MapReduce program is nothing else than a sequence $\mu_1, \rho_1, \mu_2, \rho_2, \ldots, \mu_R, \rho_R$ of map functions μ_r and reduce functions ρ_r. The input of this program is a multiset U_0 of key-value pairs. For each $r \in \{1, \ldots, R\}$, a map step, a shuffle step and a reduce step are successively executed as follows:

1. **Map step:** Each pair $\langle k; v\rangle$ in U_{r-1} is given as input to an arbitrary instance of the map function μ_r, which then produces a finite sequence of pairs. The multiset of all produced pairs is denoted by V_r.
2. **Shuffle step:** For each key k, let $V_{k,r}$ be the multiset of all values v_i such that $\langle k, v_i\rangle$. The MapReduce system automatically constructs the multiset $V_{k,r}$ from V_r in the background.
3. **Reduce step:** For each key k, a reducer (i.e., an instance calculating the reduce function ρ_r) receives k and the elements of $V_{k,r}$ in arbitrary order. We usually write such an input as a set of key-value pairs that all have key k. The reducer calculates, for each key k independently, from $V_{k,r}$ a set $U_{k,r}$ of key-value pairs. The output will then consist of all key-value pairs computed in this reduce step; that is, U_r is the union over all sets $U_{k,r}$.

Fix any ε with $0 < \varepsilon \leq 1/2$ and denote the size of the MapReduce program's input by N. For every $i \in \mathbb{N}$, a problem is in $\mathcal{MRC}^i$ if and only if if there is a MapReduce program $\mu_1, \rho_1, \mu_2, \rho_2, \ldots, \mu_R, \rho_R$ satisfying the following properties:

1. It outputs a correct answer to the problem with probability at least 3/4.
2. The number of rounds of the MapReduce program, R, is in $O(\log^i N)$.
3. The potentially randomized primitives (i.e., all map and reduce functions) are computable by a RAM with $O(\log N)$-bit words using $O(N^{1-\varepsilon})$ space and time polynomial in N.
4. The pairs produced by the map functions can be stored in $O(N^{2(1-\varepsilon)})$ space.

A MapReduce program satisfying these conditions is called an $\mathcal{MRC}^i$-algorithm. Note that due to the last condition it is impossible to even store the input unless $2(1-\varepsilon) \geq 1$, which explains the restriction $0 < \varepsilon \leq 1/2$. As with $\mathcal{NC}$, we define the union class $\mathcal{MRC} = \bigcup_{i=0}^{\infty} \mathcal{MRC}^i$. Requiring all primitives to be deterministic yields the analogous hierarchy of $\mathcal{DMRC} = \bigcup_{i=0}^{\infty} \mathcal{DMRC}^i$. Note that we obviously have $\mathcal{DMRC}^i \subseteq \mathcal{MRC}^i$ for all $i \in \mathbb{N}$. We will often refer to the single rounds of such MapReduce algorithms as $\mathcal{MRC}$-rounds and $\mathcal{DMRC}$-rounds, respectively.

3 Simulating Parallel Computations in MapReduce

We are now going to prove our two main results $\mathcal{NC}^1 \subseteq \mathcal{DMRC}^0$ for $0 < \varepsilon \leq 1/2$ and $\mathcal{NC}^{i+1} \subseteq \mathcal{DMRC}^i$ for all $i \in \mathbb{N}_+$ and $0 < \varepsilon < 1/2$ in Sections 3.2 and 3.3, respectively. In both cases, we will be making use of a technical tool derived in Section 3.1 and obtain the results by showing how to use MapReduce computations for two different, delicate simulations. For the inclusion $\mathcal{NC}^1 \subseteq \mathcal{DMRC}^0$, we simulate width-bounded branching programs that are equivalent to the respective circuits by Barrington's classical theorem [2], whereas for the higher levels of the hierarchy, we directly simulate the combinational circuits themselves.

3.1 A Technical Tool

Goodrich et al. [10] parametrize MapReduce algorithms, on the one hand, by the memory limit m for the input/output buffer of the reducers and, on the other hand, by the *communication complexity* K_r of round r, that is, the total size of inputs and outputs for all mappers and reducers in round r. We state a useful result from [10].

▶ **Theorem 1.** *Any CRCW-PRAM algorithm using M total memory, P processors and T time can be simulated in $O(T \log_m P)$ deterministic MapReduce-rounds with communication complexity $K_r \in O((M+P)\log_m(M+P))$.*

We denote by N the size of the smallest circuit representation of the CRCW-PRAM algorithm (i.e., its number of edges) plus the size of its input. Taking into account our requirements $m \in O(N^{1-\varepsilon})$ and $K_r \in O(N^{2(1-\varepsilon)})$, we obtain the following a technical tool, which will prove to be useful in our endeavor.

▶ **Corollary 2.** *Any CRCW-PRAM algorithm using M total memory, P processors and T time can be simulated in $O(T \log_{N^{1-\varepsilon}} P)$ $\mathcal{DMRC}$-rounds if $(M+P)\log_{N^{1-\varepsilon}}(M+P) \in O(N^{2(1-\varepsilon)})$.*

3.2 Simulating $\mathcal{NC}^1$

It is known that Nonuniform-$\mathcal{NC}^1$ is equal to the class of languages recognized by nonuniform width-bounded branching programs. A careful inspection of the proof due to Barrington [2] – crucially relying on the non-solvability of the permutation group on 5 elements – reveals that it naturally translates to the uniform analogue: Our uniform class $\mathcal{NC}^1$ is identical with the class of languages recognized by uniform width-bounded branching programs. In order to prove $\mathcal{NC}^1 \subseteq \mathcal{DMRC}^0$, it therefore suffices to show how to simulate such branching programs by appropriate MapReduce computations with a constant number of rounds.

We first define what a width-bounded branching program is. Let $n, w \in \mathbb{N}_+$. The input to the program is an assignment α to n Boolean variables $\mathcal{X} = \{x_0, \ldots, x_{n-1}\}$. An *instruction* or *line* of the program is a triple (x_i, f, g), where i is the *index* of an input variable $x_i \in \mathcal{X}$ and f and g are endomorphisms of $[w]$. An instruction (x_i, f, g) *evaluates* to f if $\alpha(x_i) = 1$ and to g if $\alpha(x_i) = 0$. A *width-w branching program* of length t is a sequence of instructions

(x_{i_j}, f_j, g_j) for $j \in [t]$. We also refer to the t instructions as the lines of the program. Given an assignment α to $\mathcal{X}$, a branching program B yields a function $B(\alpha)$ that is the composition of the functions to which the instructions evaluate.

To recognize a language $L \subseteq \{0,1\}^*$, we need a family $(B_n)_{n=0}^{\infty}$ of width-w branching programs with B_n taking n Boolean inputs. We say that L is recognized by B_n if there is, for each $n \in \mathbb{N}$, a set F_n of endomorphisms of $[w]$ such that for all $\alpha \in \{0,1\}^n$, $\alpha \in L$ if and only if $B_n(\alpha) \in F_n$. If f_i and g_i are automorphisms, that is, permutations of $[w]$ for all $i \in [t]$, then B_n is called a *width-w permutation branching program*, or w-PBP for short.

▶ **Theorem 3** ([2]). *If $L \in \mathcal{NC}^1$, then L is recognized by a logspace-uniform* 5-PBP *family.*

Due to Theorem 3 it is sufficient for our purposes to simulate the w-PBPs with constant w instead of the circuit families provided by the definition of $\mathcal{NC}^1$. In order to do this, we need to encode the given w-PBP and the possible assignments in the right form, namely we express them as sets of key-value pairs. A w-PBP of length t can be described as the set $\{\langle p;\, (x_{i_p}, f_p, g_p)\rangle \mid p \in [t]\}$, where we call p the *line number* of line (x_{i_p}, f_p, g_p). Similarly, an assignment $\alpha\colon \mathcal{X} \to \{0,1\}, x_i \mapsto v_i$ to the input variables $\mathcal{X} = \{x_0, x_1, \ldots, x_{n-1}\}$ is described by the set of key-value pairs $\{\langle i; (x_i, v_i)\rangle \mid i \in [n]\}$, letting the mappers divide the information by the indices of the input variables. Let N_{O} and N_{I} be the total size of the encodings of the w-PBP and the input assignment α, respectively. Let $N = N_{\mathrm{O}} + N_{\mathrm{I}}$ and let $d = \lceil N_{\mathrm{O}}^{1-\varepsilon} \rceil$ and $\ell = \lceil N_{\mathrm{O}}^{\varepsilon} \rceil$. We denote by $\div$ the integer division. For every $q \in [t \div d]$, let w-PBP$_q$ be the qth of the subprogram blocks of w-PBP of length d, that is $\{\langle p; (x_{i_p}, f_p, g_p)\rangle \mid qd \leq p \leq (q+1)d-1\}$. For ease of readability, we assume from now on without loss of generality that $d\ell = t$, so that w-PBP can be partitioned into exactly ℓ such subprograms.

For every $q \in [\ell]$, we denote by $\mathcal{X}_q$ the subset of variables from $\mathcal{X}$ appearing in the instructions of subprogram w-PBP$_q$. An assignment $\alpha_q\colon \mathcal{X}_q \to \{0,1\}$ to these variables is represented as a set of key-value pairs in the following way. Recall that the subprogram w-PBP$_q$ is a list of lines, each of which requires the assignment of a value, either 0 or 1, for exactly one variable. Let $x_{q,j}$ be the jth variable to which a value is assigned in w-PBP$_q$, let $p_{q,j}$ denote the number of the line in which this assignment occurs for the first time in w-PBP$_q$, and let $v_{q,j}$ denote the value that is assigned to $x_{q,j}$ in this line. Now, we represent α_q by $\{\langle q; (p_{q,j}, x_{q,j}, v_{q,j})\rangle \mid j \in [|\mathcal{X}_q|]\}$. Note that despite the dependence of $\mathcal{X}_q$ on q, we always have $|\mathcal{X}_q| \leq d$. Having seen how to express w-PBP, α, and both w-PBP$_q$ and α_q for all $q \in [\ell]$ as a set of key-value pairs, we are ready to state and prove the following lemma.

▶ **Lemma 4** (Proof in Appendix A [9]). *Let L be a w*-PBP*-recognized language. If the representations of w*-PBP *and, for every $q \in [\ell]$, α_q are given, then we can decide in a* 2*-round $\mathcal{DMRC}$-computation whether $\alpha \in L$ or not.*

In the following four lemmas, we show that α_q can be computed in a constant number of rounds from w-PBP and α for every $q \in [\ell]$. The challenge lies in designing an interface between the different reducers to bridge the gap between the ℓ program blocks w-PBP$_q$ and the given assignments, initially cut into ℓ block based solely on the indices of the input variables, without exceeding the memory limits. We begin with a brief overview of the four steps.

1. For each x_i, where $i \in [n]$, we compute the number of subprograms in which x_i appears, and denote this number by $\#S(x_i)$. Note that $\#S(x_i) \leq \ell$ and that $\#S(x_i)$ is the number of all those reducers for which the value assignment of x_i is generally required to compute the resulting permutations in the corresponding subprograms.

2. We compute the prefix sums of $\#S(x_i)$. For $i \in [n]$, let $y_i = \sum_{j=0}^{i} \#S(x_j)$. Note that y_i is the number of assignment triples $(p_{q,j}, x_{q,j}, v_{q,j})$ with $0 < j \leq i$ needed to compute the action of the first i subprograms and that $y_{n-1} = \sum_{q=0}^{\ell-1} |\alpha_q|$.
3. Based on the prefix sums, we will compute a *separation* of the input variables into ℓ contiguous blocks such that, for each $q \in [\ell]$, it is feasible for reducer$_q$ to produce from the qth block the input value assignments that it needs to contribute for the next step. This is nontrivial since the number of input assignments must not exceed $O(d)$ due to the memory limitation of reducer$_q$. A *separation* of the input variables $\{x_0, \ldots, x_{n-1}\}$ is a list of $\ell - 1$ *split values* $\sigma_1, \ldots, \sigma_{\ell-1}$ such that we have ℓ ordered, contiguous blocks $\{x_0, \ldots, x_{\sigma_1}\}, \{x_{\sigma_1+1}, \ldots, x_{\sigma_2}\}, \ldots, \{x_{\sigma_{\ell-1}+1}, \ldots, x_{n-1}\}$. For notational convenience, we let $\sigma_0 = -1$ and $\sigma_\ell = n - 1$. Let $\sigma_q = \max\{j \in [n] \mid y_j \leq qd\}$ for $q \in \{1, \ldots, \ell - 1\}$. Using these split values each reducer$_q$ can provide all value assignments needed for the computation of all subprograms in the next step without violating the memory limitations.
4. We compute α_q for $q \in [\ell]$ by using w-PBP, the input assignment α, and the split values.

▶ **Lemma 5** (Proof in Appendix A [9]). *Calculating $\#S(x_i)$ is in $\mathcal{DMRC}^0$. That is, for each $i \in [n]$, $\#S(x_i)$ is computable from w-PBP in a constant number of $\mathcal{DMRC}$-rounds.*

▶ **Lemma 6** (Proof in Appendix A [9]). *Computing the prefix-sums of $\#S(x_i)$ is in $\mathcal{DMRC}^0$.*

▶ **Lemma 7** (Proof in Appendix A [9]). *Each of the split values $\sigma_1, \ldots, \sigma_{\ell-1}$ can be computed in one reducer with the required prefix-sums being made available in one more $\mathcal{DMRC}$-round.*

▶ **Lemma 8** (Proof in Appendix A [9]). *Given w-PBP, α, and the split values $\sigma_0, \ldots, \sigma_\ell$, we can, for each $q \in [\ell]$, compute α_q in a constant number of $\mathcal{DMRC}$-rounds.*

We finally obtain the desired inclusion by applying Theorem 3 and Lemmas 4 through 8.

▶ **Theorem 9.** *We have $\mathcal{NC}^1 \subseteq \mathcal{DMRC}^0$.*

3.3 Simulating $\mathcal{NC}^i$ For All $i \geq 2$

For the higher levels in the hierarchy of Nick's class, we show how to simulate the involved circuits directly. We begin with a short outline of the proof.

Let $C_n = (V_n, E_n)$ be a $\mathcal{NC}^{i+1}$ circuit with an input of size n, given as a set of nodes and a set of directed edges, together with an input assignment α. The total size of C_n in bits is N_{O}, the total size of the input assignment in bits is N_{I}, and $N = N_{\mathrm{O}} + N_{\mathrm{I}}$. Note that $\mathrm{size}(C_n)$ is polynomial in n and $\mathrm{depth}(C_n) \in O(\log^i n)$. We will take the following steps to simulate the circuit C_n with deterministic MapReduce computations:

1. We compute the level of each node in C_n.
2. The nodes and edges are sorted by their level.
3. Both the circuit C_n and the input assignment α are divided equally among the reducers.
4. We split the circuit into subcircuits computable in a constant number of rounds.
5. A custom communication scheme collects and constructs the complete subcircuits.
6. The entire circuit is evaluated via evaluation of the subcircuits.

Note that equal division of C_n in the third step is very different from the split in the forth one, where the parts may differ radically in size. Great care must be taken so as to no violate any of the memory and time restrictions, necessitating the two unlike partitions. The subsequent steps then need to mediate between these dissimilar divisions. We will show that the steps (1) to (6) can be computed in $O(\log n)$, $O(1)$, $O(1)$, $O(1)$, $O(\log n)$, and $O(\mathrm{depth}(C_n)/\log n)$ rounds, respectively, yielding the desired theorem.

▶ **Theorem 10.** *We have $\mathcal{NC}^{i+1} \subseteq \mathcal{DMRC}^i$ for all $i \in \mathbb{N}_+$ and all $0 < \varepsilon < 1/2$.*

3.3.1 Computing The Levels

We begin by showing how to compute the level of each node in the circuit in $O(\log n)$ $\mathcal{DMRC}$-rounds by simulating a CRCW-PRAM algorithm. (We mention in passing that this step requires more than a constant number of rounds, which prevents us from obtaining the result for $\mathcal{NC}^1 \subseteq \mathcal{DMRC}^0$ by simulating the circuits directly; the separate approach from Subsection 3.2 via Barrington's theorem is thus required for this case.)

In [18], an algorithm is presented that computes the levels of all nodes in a directed acyclic graph and can be computed on a CREW-PRAM with $O(n + m)$ processors in $O(\log^2 m)$ time, where n and m are the numbers of nodes and edges in the graph, respectively. The first stage of this algorithm relies partly on the computation of prefix-sums, which can be computed much more efficiently when switching to a CRCW-PRAM, as we will show below. A straightforward adaptation of the analysis in [18], taking into account the maximum in-degree and out-degree and separating out the computation of prefix-sums, yields the following result.

▶ **Lemma 11.** *Let $G = (V, E)$ be a directed acyclic graph with n nodes, m edges, maximum in-degree d_{in}, and maximum out-degree d_{out}. The level of each node in G can then be computed on a CRCW-PRAM with $P \in O(m + P_{\text{P-Sum}}(O(m)))$ processors in time $T \in O((\log m) \cdot (T_{\text{P-Sum}}(O(m)) + \log \max\{d_{\mathrm{in}}, d_{\mathrm{out}}\}))$, where $P_{\text{P-Sum}}(q)$ and $T_{\text{P-Sum}}(q)$ denote, respectively, the number of processors and the computation time to compute the prefix-sums of q numbers on a CRCW-PRAM.*

In the following lemma, we aim to lower the time and memory requirements for computing prefix-sums on a CRCW-PRAM as far as possible.

▶ **Lemma 12** (Proof in Appendix A [9]). *The prefix-sums of q numbers can be computed on a CRCW-PRAM with $P \in O(q \log q)$ processors and memory $M \in O(q)$ in constant time.*

We plug in the result of Lemma 12 into Lemma 11 and then apply it to the graph C_n. Since its in-degrees and out-degrees are bounded by a constant Δ, we have $m \leq \Delta n/2 \in O(n)$. Hence we can compute the levels of the nodes of C_n on a CRCW-PRAM with $P \in O(N \log N)$ processors in time $T \in O(\log n)$. By Corollary 2, we obtain the following result.

▶ **Lemma 13** (Proof in Appendix A [9]). *Computing the levels of all nodes in C_n is in $\mathcal{DMRC}^1$.*

3.3.2 Sorting By Levels

Once the levels of all nodes are computed, each node in the circuit can be represented as $(\mathrm{level}(x_i), x_i)$. Recall that the depth of C_n is just the maximum level. Since $\mathrm{depth}(C_n) \in O(\log^k n)$ for some $k \in \mathbb{N}_+$ and the number of nodes is bounded by the number of edges, which is $\mathrm{size}(C_n) \in O(N)$, we can encode each pair $(\mathrm{level}(x_i), x_i)$ by appending to a bit string of length $\log(c_1 \log^k n)$ another one of length $\log(c_2 N)$, for appropriate constants c_1 and c_2, which results in a bit string of length $\log(cN \log^k n)$ for $c = c_1 c_2 \in \mathbb{N}$. This enables us to identify each pair $(\mathrm{level}(x_i), x_i)$ with a different bit string, which can interpreted as an integer bounded by $cN \log^k n$. We call this integer the *sorting index* of node x_i. Crucially, we chose the bit string to start with the encoding of the level. Sorting the sorting indices thus means to sort the nodes of C_n by their level. The following lemma shows how prefix-sums can be used to perform such a sort so efficiently on a CRCW-PRAM that we can apply Corollary 2 to simulate it in a constant number of $\mathcal{DMRC}$-rounds.

▶ **Lemma 14** (Proof in Appendix A [9]). *A CRCW-PRAM with $P \in O(D \log D)$ processors and memory $M \in O(D)$ can sort any subset $I \subseteq \{1, \ldots, D\}$ of integers in constant time.*

Combining Lemma 14 and Corollary 2 we obtain, by a careful analysis using $\varepsilon \neq 1/2$, the promised result.

▶ **Corollary 15** (Proof in Appendix A [9]). *Let $c \in \mathbb{N}$ and $0 < \varepsilon < 1/2$. Any set of distinct integers from $\{1, \ldots, \lceil cN \log^k n \rceil\}$ can be sorted in a constant number of $\mathcal{DMRC}$-rounds.*

Once all the nodes are sorted by their sorting index (and therefore implicitly by their level), we can enumerate them in ascending order using the sorting index j; that is, we represent each node as the key-value pair $\langle j; (\text{level}(v), v) \rangle$. Clearly, we obtain an analogous representation of the edges of the form $\langle i; ((j, (\text{level}(v), v), (j', (\text{level}(v'), v')) \rangle$, which will prove useful later on.

3.3.3 Division of Circuit And Assignment Among Reducers

As we have already seen when discussing the branching programs, an assignment α to input variables $\mathcal{X} = \{x_0, x_1, \ldots, x_{n-1}\}$ can be represented as a set $\{\langle i; (x_i, v_i) \rangle \mid i \in [n]\}$ of key-value pairs, where $\alpha(x_i) = v_i \in \{0, 1\}$.

The circuit C_n is now divided into $\ell = N_{\mathrm{O}}^{\varepsilon}$ subsets of edges according to the sorting indices and input values that are assigned to each subset as in the case of branching programs. For every $q \in [\ell]$, let $C_n^q = \{((j, \text{level}(v), v), (j', \text{level}(v'), v')) \mid qd \leq j \leq (q+1)d - 1\}$, where $d = N_{\mathrm{O}}^{1-\varepsilon}$, be the qth subset. Note that $|C_n^q| \in O(d)$. For every $q \in [\ell]$, the set of variables appearing in C_n^q is denoted as $\mathcal{X}_q$ and the assignment α_q to $\mathcal{X}_q$ is represented as $\{\langle j; x_{q,j}, v_{q,j} \rangle \mid j \in [|\alpha_q|]\}$, where $x_{q,j}$ is the jth variable that appears as an input in C_n^q, and $v_{q,j}$ is its assignment value. Just as seen in Lemma 8 for the case of a branching program, we can now, for all $q \in [\ell]$, compute α_q from C_n and α, yielding the following lemma.

▶ **Lemma 16.** *Computing α_q from C_n and α is in $\mathcal{DMRC}^0$ for every $q \in [\ell]$.*

We can therefore assume that each input node is represented by $\langle j; (\text{level}(x_{j_i}), x_{j_i}, v_{j_i}) \rangle$, a key-value pair that is computed from C_n^q and α_q for $q \in [\ell]$ in a single $\mathcal{DMRC}$-round.

3.3.4 Division Into Subcircuits By Levels

We divide $C_n = (V_n, E_n)$ into as few subcircuits as possible such that the simulation of each subcircuit is in $\mathcal{DMRC}^0$ and we can evaluate C_n by evaluating the subcircuits sequentially.

Given $v \in V_n$ and $\delta \in \mathbb{N}$, we define the *v-down-circuit* $C_\delta^{\text{down}}(v) = (V_\delta^{\text{down}}(v), E_\delta^{\text{down}}(v))$ of depth δ to be the subcircuit of C_n induced by $V_\delta^{\text{down}}(v) = \{u \mid \text{level}(v) \leq \text{level}(u) \leq \text{level}(v) + \delta, u \to^* v\}$, where $u \to^* v$ means that there is a directed path of any length (including 0) from u to v in C_n. The *v-up-circuit* $C_\delta^{\text{up}}(v) = (V_\delta^{\text{up}}(v), E_\delta^{\text{up}}(v))$ of depth δ is analogously the subcircuit of C_n induced by $V_\delta^{\text{up}}(v) = \{u \mid \text{level}(v) - \delta \leq \text{level}(u) \leq \text{level}(v), v \to^* u\}$.

When dividing C_n into subcircuits we have two conflicting goals. On the one hand, we want as few of them as possible, which implies that they have to be of great depth. On the other hand, we need to simulate them in MapReduce without exceeding the memory bounds. A depth in $\Theta(\log n)$ turns out to be the right choice. Let $s = (\gamma \log n)/\log \Delta$, where $\Delta \geq 2$ is a constant bounding the maximum degree of C_n and γ is an arbitrary constant satisfying $0 < \gamma < 1 - 2\varepsilon$. (Note that such a γ exists exactly if $\varepsilon < 1/2$.) Since a tree of depth s and maximum degree bounded by a constant Δ contains at most $\sum_{i=1}^{s} \Delta^i$ edges, its size is in $O(\Delta^s) = O(n^\gamma) \subseteq O(N^\gamma)$. Hence each reducer may contain up to $N^{1-\varepsilon}/N^\gamma$ such subcircuits without exceeding the memory constraint of $O(N^{1-\varepsilon})$; see Figure 2 in Appendix B [9]. We denote this number of allowed subcircuits per reducer by $\beta = N^{1-\varepsilon-\gamma}$.

For each $i \in [\lceil \text{depth}(C_n)/s \rceil + 1]$, we define $L_i = i \cdot s$. For every node v on level L_i – that is, with $\text{level}(v) = L_i$ – we call the v-down-circuit (v-up-circuit, resp.) of depth s an *L_i-down-circuit* (*L_i-up-circuit*, resp.). We will construct in each reducer the v-down-circuits and v-up-circuits of depth 1 of all its nodes. From those we then construct all L_i-down-circuits and L_i-up-circuits for every i. Note that we can evaluate all L_i-down-circuits if the values of the nodes of level L_{i+1} are given. The values of the nodes v of level L_{i+1} that are necessary to compute the L_i-up-circuits are then known from the L_{i+1}-down-circuits.

When the circuit C_n is divided into L_i-down-circuits, there may exist edges of C_n that are not contained in any L_i-down-circuit. If an edge $((j_u, \text{level}(u), u), (j_v, \text{level}(v), v))$ satisfies $L_{i_u} \leq \text{level}(u) \leq L_{i_u+1}$ and $L_{i_v} \leq \text{level}(v) \leq L_{i_v+1}$ for $i_u \neq i_v$, then this edge is not included in any L_{i_u}-down-circuit nor any L_{i_v}-down-circuit. We call such edges *level-jumping edges*; see Figure 3 in Appendix B [9] for an example. We would like to replace every level-jumping edge (u, v) by a path from u to v that consists only of edges that will be part of the respective L_i-down-circuits and L_i-up-circuits in the resulting, *augmented* circuit. The following lemma states that this is possible without increasing the size by too much.

▶ **Lemma 17** (Proof in Appendix A [9]). *We can subdivide the jumping edges in C_n in a way that renders the subcircuit-wise evaluation possible without increasing the size beyond $O(N)$.*

3.3.5 Construction of Subcircuits in Reducers

Having described the subcircuits on which the evaluation of the entire circuits will be based, we now need to show how to split and construct them in the ℓ different reducers. In each reducer, we start with the nodes v contained in it that satisfy $\text{level}(v) = L_i$ for any i and the associated v-down-circuits and v-up-circuits of depth 1. We then iteratively increase the depth one by one, until the full L_i-down-circuits and L_i-up-circuits of depth up to s are constructed. Note that the nodes of any level L_i and their corresponding circuits may be scattered across multiple reducers since edges were split equally among them according to their the sorting index and not depending on the level. We therefore need to carefully implement a communication scheme that allows each reducer to encode requests for missing edges required in the construction, which are then delivered to them in multiple rounds, without exceeding any of the memory or time bounds. Taking care of all these details, we obtain the following lemma.

▶ **Lemma 18** (Proof in Appendix A [9]). *Given C_n, all L_i-down-circuits and L_i-up-circuits can be constructed in $O(\log n)$ $\mathcal{DMRC}$-rounds whenever $0 < \varepsilon < 1/2$.*

3.3.6 Evaluation Via Subcircuits

The main idea in the proof of the following lemma is to compute the evaluation values subcircuit-wise, starting with the deepest ones, and then iteratively moving up the circuit in $\text{depth}(C_n)/s$ rounds, passing on the newly computed values to the right reducers, until the value of the unique output node is known.

▶ **Lemma 19.** *If all up-circuits and down-circuits are constructed in the proper reducers, C_n can be evaluated in $O(\text{depth}(C_n)/\log n)$ $\mathcal{DMRC}$-rounds.*

4 Conclusion and Research Opportunities

In a substantial improvement over all previously known results, we have shown that $\mathcal{NC}^{i+1} \subseteq \mathcal{DMRC}^i$ for all $i \in \mathbb{N}$. In the case of $\mathcal{NC}^1 \subseteq \mathcal{DMRC}^0$, we have proved this result for every feasible choice of ε in the model, that is, for $0 < \varepsilon \leq 1/2$. For $i > 0$, we have shown the result to hold for all but one value, namely $\varepsilon = 1/2$.

Achieving these two results required a detailed description of two different, delicate simulations within the MapReduce framework. For the case of $\mathcal{NC}^1$, which is particularly relevant in practice, we applied Barrington's theorem and simulated width-bounded branching programs [2], whereas we directly simulated the circuits for the higher levels of the hierarchy. We emphasize that none of the two approaches can replace the other: Barrington's theorem only gives a characterization for the first level of the $\mathcal{NC}$ hierarchy and the second approach does not even yield $\mathcal{NC}^1 \subseteq \mathcal{MRC}^0$. (Recall that $\mathcal{DMRC}$ is just the deterministic variant of $\mathcal{MRC}$, so we have $\mathcal{DMRC}^i \subseteq \mathcal{MRC}^i$ for all $i \in \mathbb{N}$.)

We would like to briefly address the small question that immediately arises from our result, namely whether it possible to extend the inclusion $\mathcal{NC}^{i+1} \subseteq \mathcal{DMRC}^i$ of Theorem 10 to the case $\varepsilon = 1/2$. Going through all involved lemmas, we see that the two reasons that our proof does not work in this corner case are the sorting of the nodes using Lemma 15 and the construction of the up-circuits and down-circuits in Lemma 18. Regarding the former, we can avoid the restriction by allowing randomization. For the latter, it is not clear that this can be achieved, however. If there was any way to construct the levels for $\varepsilon = 1/2$ as well, then Theorem 10 would immediately extend to the full range $0 < \varepsilon \leq 1/2$ of feasible choices for ε.

Besides dealing with the small issue mentioned above, the natural next step for future research is to take the complementary approach and address the reverse relationship: Having shown in this paper how to obtain efficient deterministic MapReduce algorithms for parallelizable problems, we now aim to include $\mathcal{DMRC}^i$ into $\mathcal{NC}^{i+1}$ for all $i \in \mathbb{N}$, thus finally settling the long-standing open question of how exactly the MapReduce classes correspond to the classical classes of parallel computation.

References

1 S. Arora and B. Barak. *Computational Complexity: A Modern Approach.* Cambridge University Press, 2009.

2 D.A. Barrington. Bounded-Width Polynomial-Size Branching Programs Recognize Exactly Those Languages in NC^1. *J. of Computer and System Sciences*, 38:150–164, 1989.

3 C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, and K. Olukotum. MapReduce for machine learning on multicore. In *Advances in neural information processing systems (NIPS)*, pages 281–288, 2006.

4 T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

5 J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

6 A. K. Farahat, A. Elgohary, A. Ghodsi, and M. S. Kamel. Distributed Column Subset Selection on MapReduce. In *International Conference on Data Mining (ICDM)*, pages 171–180, 2013.

7 J. Feldman, S. Muthukrishnan, A. Sidiropoulos, C. Stein, and Z. Svitkina. On Distributing Symmetric Streaming Computations. *ACM Trans. on Algorithms*, 6(4):66:1–66:15, 2010.

8 B. Fish, J. Kun, Á. D. Lelker, L. Reyzin, and G. Turán. On the Computational Complexity of MapReduce. In *International Symposium on Distributed Computing (DISC)*, pages 1–15, 2015.

9 F. Frei and K. Wada. Efficient Circuit Simulation in MapReduce. *Technical Report arXiv.org*, cs(arXiv:1907.01624):1–20, 2019. `arXiv:1907.01624`.

10 M. Goodrich, N. Sichinava, and Q. Zhang. Sorting, Searching, and Simulation in the MapReduce Framework. In *22nd Int. Symp. on Algorithms and Computation (ISAAC)*, pages 374–383, 2011.

11 S. Kamara and M. Raykova. Parallel Homomorphic Encryption. In *Financial Cryptography Workshops*, pages 213–225, 2013.

12 H. Karloff, S. Suri, and S. Vassilvitskii. A Model of Computation for MapReduce. In *21st ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 938–948, 2010.

13 R. Kumar, B. Moseley, and S. Vassilvitskii. Fast Greedy Algorithms in MapReduce and Streaming. In *ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, pages 1–10, 2013.

14 M. F. Pace. BSP vs MapReduce. In *12th Int. Conf. on Computational Science (ICCS)*, pages 246–255, 2012.

15 A. Pietracaprina, G. Pucci, M. Riondato, F. Silvestri, and E. Upfal. Space-Round Tradeoffs for MapReduce Computations. In *26th ACM Int. Conf. on Supercomputing (ICS)*, pages 235–244, 2012.

16 T. Roughgarden, S. Vassilvitskii, and J. R. Wang. Shuffles and Circuits (On Lower Bounds for Modern Parallel Computation). *Journal of the ACM (JACM)*, 65(6):41:1–66:24, 2018.

17 A. D. Sarma, F. N. Afrati, S. Salihoglu, and J. D. Ullman. Upper and Lower Bounds on the Cost of a Map-Reduce Computation. In *Proceedings of the VLDB Endowment (PVLDB)*, pages 277–288, 2013.

18 A. Tada, M. Migita, and R. Nakamura. Parallel Topological Sorting Algorithm. *J. of the Information Processing Society of Japan (IPSJ)*, 45(4):1102–1111, 2004.

19 T. White. *Hadoop: The Definitive Guide, 4th edition.* O'Reilly, 2015.

A Deferred Proofs

In this appendix, we provide all proofs that had to be deferred due to the space constraints. For the reader's convenience, we reprint all statements.

▶ **Lemma 4** (Reprint of Lemma 4 on page 7)**.** *Let L be a w-PBP-recognized language. If the representations of the w-PBP and, for every $q \in [\ell]$, α_q are given, then we can decide in a 2-round $\mathcal{DMRC}$-computation whether $\alpha \in L$ or not.*

Proof. As already described above, let w-PBP be represented by the set $\{\langle p; (x_{i_p}, f_p, g_p)\rangle \mid p \in [t]\}$ and, for every $q \in [\ell]$, the assignment α_q by $\{\langle q, (p_{q,j}, x_{q,j}, v_{q,j})\rangle \mid j \in [|\mathcal{X}_q|]\}$. Note that there are ℓ subprograms of length at most d and ℓ partial assignments that each assign values to at most one variable per line of the corresponding partial program. The total size of the input is thus in $O(d\ell) \subseteq O(N_{\mathrm{O}}) \subseteq O(N)$.

We define the first map function μ_1 by

$$\begin{aligned}\mu_1(\langle p; (x_{i_p}, f_p, g_p)\rangle) &= \{\langle\ p \div d\,;\ (p, x_{i_p}, f_p, g_p)\ \rangle\},\ \text{for each } p \in [t] \text{ and}\\ \mu_1(\langle q; (p_{q,j}, x_{q,j}, v_{q,j})\rangle) &= \{\langle\ p_{q,j} \div d\,;\ (p_{q,j}, x_{q,j}, v_{q,j})\ \rangle\}\ \text{for each } q \in [\ell], j \in [k+1].\end{aligned}$$

For any $q \in [\ell]$, there is one subprogram w-PBP$_q$ and an associated assignment set α_q. We use the map function μ_1 to find the value assignment for each variable appearing in w-PBP$_q$ and store it in a key-value pair. This pair has the key q and is thereby designated to be processed by reducer$_q$, which can calculate ρ_1, having all pairs with key q available. This function simulates, for each permutation π of $[w]$, the subprogram w-PBP$_q$ on this permutation with the received assignment and stores the resulting permutation π'. This yields a table T_q of size $w! \in O(1)$, describing the action of w-PBP$_q$ for the given assignment on all $w!$ permutations. (We mention in passing that for the first reducer$_0$ it would be sufficient to compute and store only the permutation that results from applying w-PBP$_0$ on the given assignment to the identity as the initial permutation, thus saving the time and memory necessary for the rest of the first table.) The output of ρ_1 on the qth reducer is $\langle q; T_q\rangle$.

The map function μ_2 of the second round is simple, it maps $\langle q; T_q\rangle$ to $\langle 0; (q, T_q)\rangle$, thus delivering all pairs (i, T_i) to a single instance of the reduce function ρ_2. This first reducer has therefore all tables $T_0, \dots, T_{\ell-1}$ at its disposal and knows which one is which. Using T_q as a look-up table for the permutation performed by w-PBP$_q$, reducer$_0$ can now compute, starting from the identity permutation id, the permutation $\pi = T_{\ell-1} \circ \cdots \circ T_2 \circ T_1 \circ T_0(\text{id})$, and the input is accepted if and only if $\pi \in F_n$, where F_n is the set of accepted permutations that is given to us alongside the program w-PBP. ◀

▶ **Lemma 5** (Reprint of Lemma 5 on page 8). *Calculating $\#S(x_i)$ is in $\mathcal{DMRC}^0$. That is, for each $i \in [n]$, $\#S(x_i)$ is computable from w-*PBP *in a constant number of $\mathcal{DMRC}$-rounds.*

Proof. For each $q \in [\ell]$, the subprogram w-PBP$_q$ is stored in reducer$_q$. The output of reducer$_q$ – which will be the input to compute $\#S(x_i)$ – is $\langle q; (q, 1)\rangle, \dots, \langle q; (q, k_q)\rangle$, with the variables $x_{q,1}, \dots, x_{q,k_q}$ appearing in the subprogram w-PBP$_q$ and $k_q \in O(d)$. The total number of inputs used to compute $\#S(x_i)$ is therefore at most $d\ell \in O(N)$. We use a Sum-CRCW-PRAM, whose concurrent writes to a single memory register are resolved by summing up all values being written to the same register simultaneously, see [10]. We use at most $d\ell$ processors, $\mathrm{P}_{q,1}, \dots, \mathrm{P}_{q,k_q}$ for each $q \in [\ell]$, and registers $\mathrm{R}_0, \dots, \mathrm{R}_{n-1}$ and let all processors $\mathrm{P}_{q,j}$ add 1 to R_j concurrently. Thus we see that computing $\#S(x_i)$ is possible in constant time on a Sum-CRCW-PRAM and therefore, by Corollary 2, in $\mathcal{DMRC}^0$. ◀

▶ **Lemma 6** (Reprint of Lemma 6 on page 8). *Computing the prefix-sums of $\#S(x_i)$ is in $\mathcal{DMRC}^0$.*

Proof. The input is given as $\langle i; (\#S(x_i), i)\rangle$ for $i \in [n]$. We compute the prefix-sums y_i of $\#S(x_i)$ for all $i \in [n]$ in three rounds that can be summarized as follows:

1. Each reducer$_q$, for $q \in [\ell]$, determines its local prefix-sums; that is, it computes the d prefix-sums $y^{\text{local}}_{dq}, \dots, y^{\text{local}}_{d(q+1)-1}$ of the d numbers $\#S(x_{dq}), \dots, \#S(x_{d(q+1)-1})$.
2. A single reducer computes the prefix-sums $z_0, z_1, \dots z_{\ell-1}$ of $y^{\text{local}}_{d-1}, y^{\text{local}}_{2d-1}, \dots y^{\text{local}}_{\ell d-1}$, which are known from the first round. For every $q \in [\ell - 1]$, we send z_q to reducer$_{q+1}$.
3. Each reducer$_{q+1}$ with $q \in [\ell - 1]$ computes $y_{d(q+1)+j} = y^{\text{local}}_{d(q+1)+j} + z_q$ for each $j \in [d]$.

We now describe the three rounds in more detail at the level of the key-value pairs.

1. By defining the map function $\mu_1(\langle i; (\#S(x_i), i)\rangle) = \langle i \div d; (\#S(x_i), i)\rangle$, each reducer$_q$, for $q \in [\ell]$, receives $\#S(x_{dq}), \dots, \#S(x_{d(q+1)-1})$ together with the correct indices. Thus we can compute in reducer$_q$ all local prefix-sums $y^{\text{local}}_{dq}, \dots, y^{\text{local}}_{d(q+1)-1}$ of these number. The output of reducer$_q$ consists of the local prefix-sums in the format $\langle q; (\text{p-sum}, q, j, y^{\text{local}}_{q,j})\rangle$ for $j \in [d]$ and the last of each group of local prefix-sums in the format $\langle q; (\text{last}, y^{\text{local}}_{d(q+1)-1})\rangle$, where p-sum $= 0$ and last $= 1$ is a simple binary identifier.
2. By defining the map function $\mu_2(\langle q; (\text{last}, y^{\text{local}}_{d(q+1)-1})\rangle) = \langle 0; (\text{last}, y^{\text{local}}_{d(q+1)-1})\rangle$, all last parts of the local prefix-sums can be gathered in reducer$_0$. Thus, the prefix-sums $z_0, z_1, \dots z_{\ell-1}$ of $y^{\text{local}}_{d-1}, \dots, y^{\text{local}}_{d\ell-1}$ can be computed in it and the output of the reducer is $\langle 0; (\text{last}, i + 1, z_i)\rangle$ for every $i \in [\ell - 1]$. All other key-value pairs – that is, those of the form $\langle q; (\text{p-sum}, q, j, y^{\text{local}}_{q,j})\rangle$ – are passed on unaltered.
3. The input of the third round consists of the output pairs $\langle q; (\text{p-sum}, q, j, y^{\text{local}}_{q,j})\rangle$ for all $j \in [d]$ and $q \in [\ell]$ passed on from the first round and the pairs $\langle 0; (\text{last}, q+1, z_q)\rangle$ for all $q \in [\ell - 1]$ from the second round. Defining the map function as $\mu_3(\langle q; (\text{p-sum}, q, j, y^{\text{local}}_{q,j})\rangle) =$

$\langle q; (\text{p-sum}, q, j, y^{\text{local}}_{q,j})\rangle$ and $\mu_3(\langle 0; (\text{last}, q+1, z_q)\rangle) = \langle q+1; (\text{last}, q+1, z_q)\rangle$, we can, for each $j \in [d]$ and each $q \in \{1, \ldots, \ell-1\}$, compute $y_{q,j} = y^{\text{local}}_{q,j} + z_j$ in reducer_q.

The memory limitations of the mappers and reducers are clearly respected. ◀

▶ **Lemma 7** (Reprint of Lemma 7 on page 8). *Each of the split values $\sigma_1, \ldots, \sigma_{\ell-1}$ can be computed in one reducer with the required prefix-sums being made available in one more $\mathcal{DMRC}$-round.*

Proof. If there is a $k \in [\ell-1]$ such that $y_{n-1} \le kd$, then it is clear from the definition $\sigma_q = \max\{j \in [n] \mid y_j \le qd\}$ of the split values that $\sigma_k = \sigma_{k+1} = \ldots = \sigma_{\ell-1}$. We can therefore assume that $y_{n-1} > (\ell-1)d$ and characterize, for each $q \in \{1, \ldots, \ell-1\}$, the split value σ_q as the unique integer satisfying $(q-1)d < y_{\sigma_q} \le qd$ and $qd < y_{\sigma_q+1}$; see Figure 1 in Appendix B [9].

This characterization is well defined since $0 < \#S(x_i) \le \ell < d$ for each $i \in [n]$ and $y_{n-1} \le d\ell \in O(N_{\text{O}})$. For each $q \in [\ell]$, in order to determine the split value σ_q, it is therefore sufficient to have available in the respective reducer a sequence of consecutive prefix-sums such that the first one is at most qd and the last one is greater than qd. This condition is satisfied if reducer_q has the $d+2$ consecutive prefix-sums $y_{qd-1}, y_{qd}, \ldots, y_{(q+1)d-1}, y_{(q+1)d}$ available. (For the first and the last reducer, the $d+1$ prefix-sums $y_0, \ldots, y_{d-1}, y_d$ and $y_{(\ell-1)d-1}, y_{(\ell-1)d}, \ldots, y_{\ell d-1}$, respectively, will suffice.) Slightly extending the sequence of available prefix-sums in each reducer by copying the overlapping prefix-sums from another reducer thus enables us to compute all split values in the ℓ reducers. Since for each $q \in [\ell]$, there are the d prefix-sums $y_{qd}, \ldots, y_{(q+1)d-1}$ in reducer_q, each reducer can have the $d+2$ prefix-sums made available after one more round by having each neighboring reducer copy one more prefix-sum into it. We have $\sigma_0 = -1$ and $\sigma_\ell = n-1$; it is thus immediately verified that, for every $q \in [\ell]$, the total number of subprograms in which input variables between x_{σ_q+1} and $x_{\sigma_{(q+1)}}$ appear is at most $2d$, showing that all the memory restrictions on the reducers are observed. ◀

▶ **Lemma 8** (Reprint of Lemma 8 on page 8). *Given w-PBP, α, and the split values $\sigma_0, \ldots, \sigma_\ell$, we can, for each $q \in [\ell]$, compute α_q in a constant number of $\mathcal{DMRC}$-rounds.*

Proof. We can assume that, for each $\kappa \in [\ell]$, the reducer_κ has the subprogram $w\text{-PBP}_\kappa$, the κth block of input assignments $\{(x_j, v_j) \mid \kappa \cdot d \le j \le (\kappa+1)d - 1\}$, and the split values $\sigma_0, \ldots, \sigma_\ell$ available. The output of reducer_κ then consists of the following:

1. $\langle \kappa; (q, p, x_{i_p}, f_p, g_p)\rangle$ for each line (p, x_{i_p}, f_p, g_p) in $w\text{-PBP}_\kappa$, where $\sigma_q + 1 \le i_p \le \sigma_{q+1}$.
2. $\langle \kappa; (q, x_j, v_j)\rangle$ for each value assignment (x_j, v_j) with $\sigma_q + 1 \le j \le \sigma_{q+1}$.

For any $\kappa \in [\ell]$, we need to bound the total number of outputs with key κ from above. From the definition of the split values we see that this number is in $O(d)$ since it is bounded by the number of lines, which is at most $2d$, plus the number of assignments, which is at most d.

Naturally, the map function μ of the next round is defined by

1. $\mu(\langle \kappa; (q, p, x_{j_p}, f_p, g_p)\rangle) = \langle q; (p, x_{j_p}, f_p, g_p)\rangle$ and
2. $\mu(\langle \kappa; (q, x_j, v_j)\rangle) = \langle q; (x_j, v_j)\rangle$.

For any $\kappa \in [\ell]$, the assignment variables α_q can be computed by the subsequent reduce function using the key-value pairs produced above. For each $q \in [\ell]$, the reducer_q has now available the lines of w-PBP and the value assignments for the input variables between x_{σ_q+1} and $x_{\sigma_{q+1}}$. It can therefore go through all the program lines and determine, on the one hand, which value assignments they require and, on the other hand, to which subprogram they belong. The required assignment information is then sent to the respective reducers by outputting $\langle q; (p \div d, p, x_{i_p}, v_{i_p})\rangle$. ◀

▶ **Lemma 12** (Reprint of Lemma 12 on page 9)**.** *The prefix-sums of q numbers can be computed on a CRCW-PRAM with $P \in O(q \log q)$ processors and memory $M \in O(q)$ in constant time.*

Proof. We use a Sum-CRCW-PRAM, where concurrent writes to the same memory register are resolved by adding up all simultaneously assigned numbers [10]. Let q numbers $x_0, x_1, \dots, x_{q-1}$ be given as input. Without loss of generality, we assume q to be a power of 2 and calculate $s_i(j) = \sum_{j2^i \le p < (j+1)2^i} x_p$ for all $i \in [1 + \log q]$ and all $j \in [q/2^i + 1]$; see Figure 4 in Appendix B [9] for an illustrating example.

Since each of the $q/2^i$ elements in s_i is the sum of 2^i elements, we can – by allocating q processors for each $i \in [1 + \log q]$ – compute every $s_i(j)$ in a Sum-CRCW-PRAM with $O(q \log q)$ processors and $O(1)$ time.

We now describe how the prefix-sums $y(0), y(1), \dots, y(q-1)$ are computed from the $s_i(j)$. Assume first that $j + 1$ is a power of 2, that is, $j + 1 = 2^p$. Then we have $y(j) = s_p(0)$, so the value has already been computed. If $j + 1 = 2^p + 1$ for some p, then we have $y(j) = s_p(0) + s_0(2^p)$, so we need to add two summands. In general, $y(j)$ can be calculated as the sum of at most $\log q - 1$ known summands.

Let $a^j_{\log q} a^j_{(\log q)-1} \dots a^j_0$ be the binary representation of $j + 1$. Now, we can see that

$$\begin{aligned} y(j) = {} & s_{\log q}(0) \cdot a^j_{\log q} \\ & + s_{(\log q)-1}((j + 1 - 2^{(\log q)-1}) \div 2^{(\log q)-1}) \cdot a^j_{(\log q)-1} \\ & + \dots \\ & + s_1((j + 1 - 2^1) \div 2^1) \cdot a^j_1 \\ & + s_0((j + 1 - 2^0) \div 2^0) \cdot a^j_0; \end{aligned}$$

that is, $y(j)$ can be computed as the sum of all $s_p((j + 1 - 2^p) \div 2^p)$ such that $a^j_p = 1$. Thus, it is sufficient to supply a maximum of $(\log q) - 1$ processors for the calculation of each $y(j)$ in a second time step, and the prefix-sums can be computed on a Sum-CRCW-PRAM with $O(q \log q)$ processors in constant time. ◀

▶ **Lemma 13** (Reprint of Lemma 13 on page 9)**.** *Computing the levels of all nodes in C_n is in $\mathcal{DMRC}^1$.*

Proof. From Lemmas 11 and 12 we know that the level of each node in C_n can be computed in $T \in O(\log n)$ time on a Sum-CRCW-PRAM with $P \in O(N + N \log N)$ processors. Now, Corollary 2 yields a MapReduce simulation of this Sum-CRCW-PRAM. We need to check that the conditions of Corollary 2 are indeed all satisfied: From $T \in O(\log n)$, $P \in O(N + N \log N)$, and $M \in O(N)$ follows $M + P \in O(N \log N)$ and $\log_{N^{1-\varepsilon}}(M + P) \in O(1)$, hence we have $(M + P) \log_{N^{1-\varepsilon}}(M + P) \in O(N^{2(1-\varepsilon)})$. Thus, the level of each node in C_n can be computed in $O(\log n)$ $\mathcal{DMRC}$-rounds. ◀

▶ **Lemma 14** (Reprint of Lemma 14 on page 10)**.** *A CRCW-PRAM with $P \in O(D \log D)$ processors and memory $M \in O(D)$ can sort any subset $I \subseteq \{1, \dots, D\}$ of integers in constant time.*

Proof. Recall that we use a Sum-CRCW-PRAM that sums up concurrent writes. Assume that the input and output are stored in the arrays $x[0], \dots, x[p-1]$ and $y[0], \dots, y[p-1]$, respectively. We will use two auxiliary arrays $z[0], \dots, z[D]$ and $\hat{z}[0], \dots, \hat{z}[D]$ of size $D + 1$. The algorithm works in four steps:

1. Initialize z by using $D+1 \leq P$ processors to set $z[k] \leftarrow 0$ for all $k \in [D+1]$.
2. Use $p \leq P$ processors in parallel to set $z[x[k]] \leftarrow 1$ for all $k \in [p]$.
3. Compute the prefix-sums of the array z and save them into $\hat{z}$.
4. Use D processors to set, for all $k \in \{1, \dots, D\}$ in parallel, $y[\hat{z}[k]] \leftarrow k$ if and only if $\hat{z}[k] \neq \hat{z}[k-1]$.

Since the prefix-sums of D numbers can be computed by the Sum-CRCW PRAM with $P \in O(D \log D)$ processors and memory $M \in O(D)$ in constant time by Lemma 12, the above algorithm stays within these bounds as well.

We now prove that this algorithm is correct. First we observe that after step 2, for every $k \in \{1, \dots, D\}$, we have $z[k] = 1$ if and only if one of the p integers to be sorted is k. Because $\hat{z}$ contains the prefix-sums of z, the value stored in $\hat{z}[k]$ hence tells us how many of the p integers in x are at most k. (Note that accordingly we always have $z[0] = \hat{z}[0] = 0$.) Thus k is one of the integers in x if and only if $\hat{z}[k] = \hat{z}[k-1] + 1$; otherwise, we have $\hat{z}[k] = \hat{z}[k-1]$. As a consequence, the array $\hat{z}$ contains exactly the indices of x, namely $[p]$, as values in non-decreasing order, that is, $0 = \hat{z}[0] \leq \hat{z}[1] \leq \cdots \leq \hat{z}[D-1] \leq \hat{z}[D] = p$. Stepping through $\hat{z}$ from start to end, that is, from $k = 0$ to $k = D$, we therefore observe an increment of 1 from $\hat{z}[k-1]$ to $\hat{z}[k]$ exactly if k is one of the integers to be sorted. This means that in step 4 the integers contained in x are detected from left to right in ascending order and subsequently stored into y in the same order. ◀

▶ **Corollary 15** (Reprint of Lemma 15 on page 10). *Let $c \in \mathbb{N}$ and $0 < \varepsilon < 1/2$. Any set of distinct integers from $\{1, \dots, \lceil cN \log^k n \rceil\}$ can be sorted in a constant number of $\mathcal{DMRC}$-rounds.*

Proof. We apply Lemma 14 with $D \in O(N \log^k n)$. We have $D \in O(N \log^k N) \subseteq O(N^{1+\zeta})$ and thus also $D \log D \in O(N^{1+\zeta})$ for any constant $\zeta > 0$. Choose any $\zeta < 1 - 2\varepsilon$, which is possible for $\varepsilon < 1/2$. The sorting is then possible on a CRCW-PRAM with $O(N^{1+\zeta})$ processors and $O(N^{1+\zeta})$ memory in constant time. By Corollary 2, this CRCW-PRAM can be simulated in a constant number of $\mathcal{DMRC}$-rounds because $\log_{N^{1-\varepsilon}}(N^{1+\zeta}) = (1+\zeta)/(1-\varepsilon) \in O(1)$ and $O(N^{1+\zeta}) \subseteq O(N^{2(1-\varepsilon)})$. ◀

▶ **Lemma 17** (Reprint of Lemma 17 on page 11). *We can subdivide the jumping edges in C_n in a way that renders the subcircuit-wise evaluation possible without increasing the size beyond $O(N)$.*

Proof. Let $((j_u, \mathrm{level}(u), u), (j_v, \mathrm{level}(v), v))$ be a jumping edge, where $L_{i_u} \leq \mathrm{level}(u) \leq L_{i_u+1}$, $L_{i_v} \leq \mathrm{level}(v) \leq L_{i_v+1}$, and $i_u < i_v$. If $i_u = i_v - 1$, then this edge is divided into two edges $((j_u, \mathrm{level}(u), u), \mathrm{dummy})$ and $(\mathrm{dummy}, (j_v, \mathrm{level}(v), v))$, introducing a new node dummy of the id kind with $\mathrm{level}(\mathrm{dummy}) = i_v$. If $i_u \leq i_v - 2$, then this edge is divided into three edges $((j_u, \mathrm{level}(u), u), \mathrm{dummy}_1)$, $(\mathrm{dummy}_1, \mathrm{dummy}_2)$, and $(\mathrm{dummy}_2, (j_v, \mathrm{level}(v), v))$, introducing two new nodes with $\mathrm{level}(\mathrm{dummy}_1) = i_u + 1$, $\mathrm{level}(\mathrm{dummy}_2) = i_v$. Having divided the jumping edges in this way, the newly created edges are all part of some dummy-down-circuit or dummy-up-circuit, except for edges of the form $(\mathrm{dummy}_1, \mathrm{dummy}_2)$. Note that we cannot further subdivide the edges of the form $(\mathrm{dummy}_1, \mathrm{dummy}_2)$ because we would exceed the size limit on the circuit otherwise. The most convenient way to deal with this is to adjust our definition of down-circuits and up-circuits such that every edge of the form $(\mathrm{dummy}_1, \mathrm{dummy}_2)$ is considered to be both a dummy_1-down-circuit and a dummy_2-up-circuit on its own. This way, every edge in the augmented circuit is included in some down-circuit or up-circuit. Note that this augmentation can be performed in a single round and that the size of the augmented circuit is in $O(N)$. In what follows, we consider C_n to be this augmented circuit. ◀

▶ **Lemma 18** (Reprint of Lemma 18 on page 11). *Given C_n, all L_i-down-circuits and L_i-up-circuits can be constructed in $O(\log n)$ $\mathcal{DMRC}$-rounds whenever $0 < \varepsilon < 1/2$.*

Proof. In the first round, the map function μ_1 is defined such that each reducer$_q$ is assigned (via the choice of the key) β nodes of the form $\langle j; (\text{level}(v), v)\rangle$ and directed edges adjacent to these nodes. Note that one edge can thus be assigned to two different reducers, once as an outgoing and once as an incoming edge. Specifically, we define

$$\mu_1(\langle\, j\, ;\, (\text{level}(v), v)\,\rangle) = \{\langle\, j \div \beta\, ;\, (j, \text{level}(v), v)\,\rangle\}$$

for the key-value pairs representing nodes and

$$\begin{aligned}\mu_1(\langle i; (\, (j, \text{level}(v), v), (j', \text{level}(v'), v')\,)\rangle) = \{\ &\langle j \div \beta;\ ((j, \text{level}(v), v), (j', \text{level}(v'), v'))\rangle,\\ &\langle j' \div \beta;\ ((j, \text{level}(v), v), (j', \text{level}(v'), v'))\rangle\ \}\end{aligned}$$

for the key-value pairs representing edges.

In the subsequent execution of ρ_1, each reducer can therefore directly construct the v-up-circuits and v-down-circuits of depth 1 for its β assigned nodes. We will now describe how some of these initial circuits, namely those on levels L_i for any $i \in [r]$, can be extended to full L_i-up-circuits and L_i-down-circuits by iteratively increasing the circuit depth one by one in the following way:

Let v be a node with $\text{level}(v) = L_i$ in reducer$_q$ for any $i \in [r]$ and $q \in [\ell]$. We want to extend $C_1^{\text{down}}(v)$ and $C_1^{\text{up}}(v)$ to $C_2^{\text{down}}(v)$ and $C_2^{\text{up}}(v)$, respectively. Let u_{in} (u_{out}, resp.) be any node of in-degree (out-degree, resp.) 0 in it, that is, any node that potentially needs to be extended by one or multiple edges. These extending edges are not necessarily available in reducer$_q$, however. We need to find out which reducer stores them – if there are any – and then request these edges from it in some way. To determine the right reducer, we make use of the sorting index stored alongside each node, even when part of an edge. Any edge (u_{in}, v) that we need to check for possible extensions is in fact represented as $\langle q\ ,\ (\,(j_{u_{\text{in}}}, \text{level}(u_{\text{in}}), u_{\text{in}})\,,\,(j_v, \text{level}(v), v)\,)\ \rangle$ in reducer$_q$. The number of the reducer containing the downward extending edges is now retrieved as $\text{to}(u_{\text{in}}) = j_{u_{\text{in}}} \div \beta$. Analogously, the upward extending edges for an edge (v, u_{out}) are to be found in reducer$_{\text{to}(u_{\text{out}})}$, where $\text{to}(u_{\text{out}}) = j_{u_{\text{out}}} \div \beta$. We now know whom to ask for edges extending the subcircuit beyond node u, namely reducer number $\text{to}(u)$. Let $\text{from}(v) = q$ denote the number of the reducer sending the request, which we encode in form of the key-value pair $\langle q; (u, \text{to}(u), \text{from}(v))\rangle$.

Each reducer$_q$ does the above for every node with possible extending edges and also passes along to the mapper all v-up-circuits and v-down-circuits constructed so far unaltered. This concludes the first round. In the second round, the map function μ_2 naturally reassigns $\langle q; (u, \text{to}(u), \text{from}(v))\rangle$ to reducer$_{\text{to}(u)}$, and returns the v-up-circuits and v-down-circuits to the reducers that sent them. Having received the edge request of the form $\langle \text{to}(u); (u, \text{to}(u), \text{from}(v))\rangle$ while executing ρ_2, reducer$_{\text{to}(u)}$ now sends all edges potentially useful to reducer$_{\text{from}(v)}$ – that is, the entire u-up-circuit and the entire u-down-circuit of depth 1 – to the next mapper in the form of a pair $(\text{from}(v), e)$ for every edge containing node u. As before, all other circuits constructed so far get passed along without modification as well.

In the third round, the map function μ_3 routes the requested edges to the requesting reducer by generating the key-value pairs $\langle \text{from}(v); (\text{from}(v), e)\rangle$. In the reducing step, which implements the same reduce function ρ_1 as in the first round, $\text{reducer}_{\text{from}(v)}$ now finally has all v-up-circuits and v-down-circuits fully extended to depth 2.

Since performing the two rounds $\mu_2, \rho_2, \mu_3, \rho_1$ deepens the L_i-up-circuits and L_i-down-circuits by one level in the way just seen, the complete L_i-up-circuits and L_i-down-circuits can be constructed by repeating these two rounds s times.

It is again clear that the memory and I/O requirements of the reducers are all met in every round since the input size and output size are in $O(d)$ for each reducer. Moreover, the total memory for storing the v-up-circuits and v-down-circuits is $\beta \cdot N \in O(N^{1+\gamma})$ because C_n has at most $N_{\text{O}} \in O(N)$ nodes. Since the constant γ was chosen such that $0 < \gamma \leq 1 - 2\varepsilon$, we have $N^{1+\gamma} \in O(N^{2(1-\varepsilon)})$ and thus all up-circuits and down-circuits can be stored in the respective reducers. ◀

▶ **Lemma 19** (Reprint of Lemma 19 on page 11). *If all up-circuits and down-circuits are constructed in the proper reducers, C_n can be evaluated in $O(\text{depth}(C_n)/\log n)$ $\mathcal{DMRC}$-rounds.*

Proof. Without loss of generality, let $\text{depth}(C_n)$ be divisible by s and let $r = \text{depth}(C_n)/s$. Once all L_i-down-circuits and L_i-up-circuits for all $i \in \{1, \ldots, r\}$ have been constructed, we can evaluate C_n on the given input assignment. We begin by evaluating the L_{r-1}-down-circuits. Since every input node has its value assigned in a v-down-circuit, the L_{r-1}-down-circuits can be computed in the reducers containing these v-down-circuits. With the values of all nodes at level L_{r-1} determined, we can send the necessary values to the L_{r-2}-down-circuits and, in the case of edges that were divided using two dummy nodes, to lower-level down-circuits. Nodes at level L_{r-1} that are necessary to compute L_{r-2}-down-circuits are described in the L_{r-1}-up-circuits. Any node v at level L_{r-1} that is necessary to compute L_{r-2}-down-circuits is described in the v-up-circuit. Therefore, the output of the reducer_q is as follows: Let v be at level L_{r-1} and let u_i, for $i \in \{1, \ldots, k_v\}$, be the nodes at level L_{r-2} in the v-up-circuit. For each v in reducer_q, it outputs $(\text{to}(u_i), v, \text{val}(v))$, where $\text{to}(u_i)$ is the index of the reducer containing the u_i-down-circuit and $\text{val}(v)$ is the value of v determined in the computation of the v-down-circuit. The reducer_q also passes on all v-down-circuits and v-up-circuits contained in it.

In the next round, the map function sends each $(\text{to}(u_{\text{in}}), v, \text{val}(v))$ to the reducer containing the u_{in}-down-circuit; that is, it generates the key-value pair $\langle \text{to}(u_{\text{in}}); (v, \text{val}(v))\rangle$. Of course, the map function also passes along all v-down-circuits and v-up-circuits to the proper reducers.

Since now each L_{r-2}-down-circuit is contained completely in a reducer that has gathered all values of nodes at level L_{r-1} necessary to compute this subcircuit, all L_{r-2}-down-circuits can be computed in their reducers. Now we can compute the values of nodes higher and higher up in the circuit, by iterating the last mapping-reducing function pair, until the value is finally known for the unique output node. As before, we clearly stay within the memory and I/O buffer limits of each reducer. ◀

B Illustrating Figures

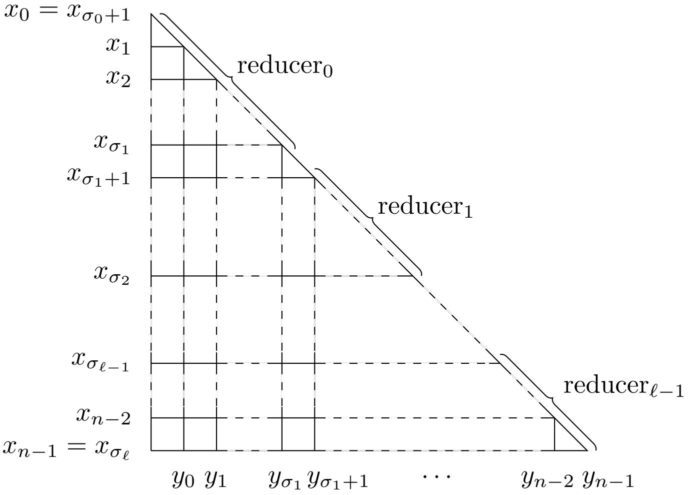

Figure 1 Separation of the input variables $x_0, \ldots, x_{n-1}$ into ℓ blocks for the ℓ reducers, in dependence of the values of y_i.

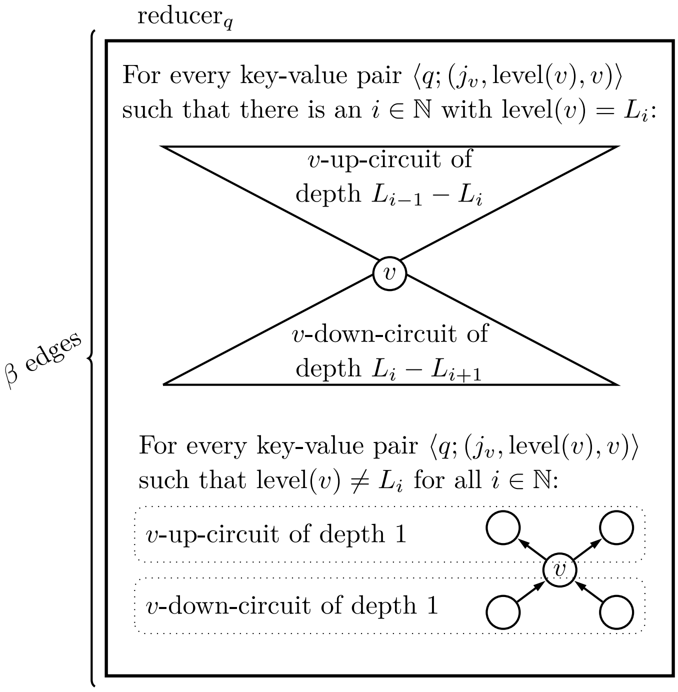
reducer$_q$
For every key-value pair $\langle q; (j_v, \text{level}(v), v)\rangle$
such that there is an $i \in \mathbb{N}$ with level$(v) = L_i$:
v-up-circuit of
depth $L_{i-1} - L_i$
v
v-down-circuit of
depth $L_i - L_{i+1}$
For every key-value pair $\langle q; (j_v, \text{level}(v), v)\rangle$
such that level$(v) \neq L_i$ for all $i \in \mathbb{N}$:
v-up-circuit of depth 1
v-down-circuit of depth 1
v
β edges

Figure 2 The up-circuits and down-circuits constructed in reducer$_q$, comprising up to β edges.

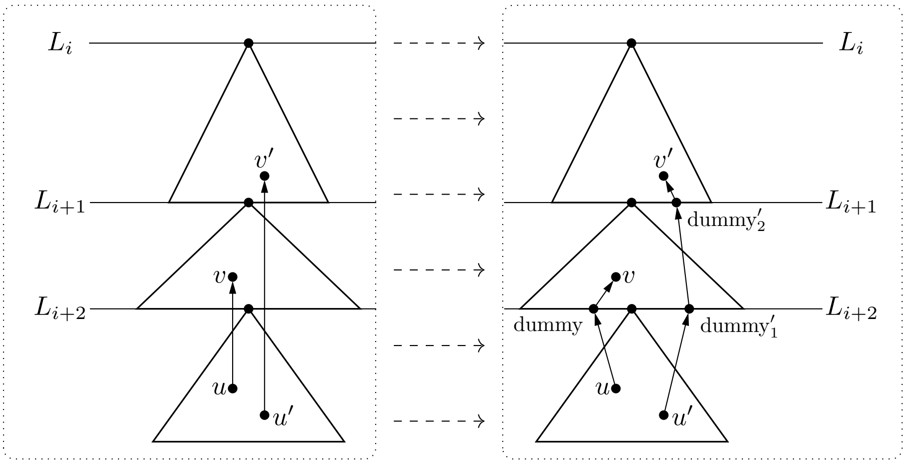

Figure 3 Two jumping edges on the left and their resolving division on the right.

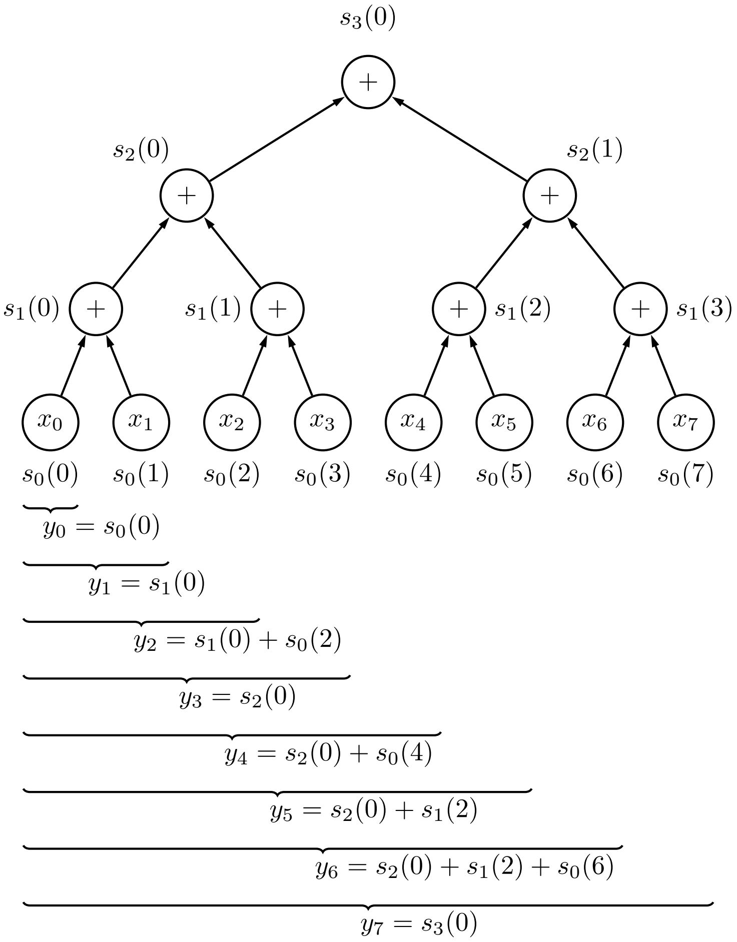

Figure 4 Calculation of the prefix-sums $s_i(j) = \sum_{p \in [(j+1)2^i] \setminus [j2^i]} x_p$ for every $i \in [1 + \log q]$ and $j \in [q/2^i]$ for the example of $q = 8$.