A Theory of Partitioned Global Address Spaces*

Georgel Calin[1], Egor Derevenetc[2], Rupak Majumdar[3], and Roland Meyer[1]

1 **University of Kaiserslautern, Germany,** {calin,meyer}@cs.uni-kl.de
2 **Fraunhofer ITWM, Germany,** egor.derevenetc@itwm.fraunhofer.de
3 **MPI-SWS, Germany,** rupak@mpi-sws.org

Abstract

Partitioned global address space (PGAS) is a parallel programming model for the development of high-performance applications on clusters. It provides a global address space partitioned among the cluster nodes, and is supported in programming languages like C, C++, and Fortran by means of APIs. Our first contribution is a formal model for the semantics of single program, multiple data programs that use PGAS APIs. Our model reflects the main features of popular real-world APIs such as SHMEM, ARMCI, GASNet, GPI, and GASPI.

A key feature of PGAS is the support for one-sided communication: a node may directly read and write the memory located at a remote node, without explicit synchronization with the processes running on the remote side. One-sided communication increases performance by decoupling process synchronization from data transfer, but requires the programmer to reason about appropriate synchronizations between reads and writes. As a second contribution, we propose and investigate *robustness*, a criterion for correct synchronization of PGAS programs. Robustness corresponds to acyclicity of a suitable happens-before relation defined on PGAS computations. The requirement is finer than classical data race freedom and rules out most false error reports.

Our main technical result is an algorithm for checking robustness of PGAS programs. The algorithm makes use of two insights. We first show that, if a PGAS program is not robust, then there are computations in a certain normal form that violate happens-before acyclicity. Intuitively, normal-form computations delay remote accesses in an ordered way. We then devise an algorithm that checks for cyclic normal-form computations. Essentially, the algorithm is an emptiness check for a novel automaton model that accepts normal-form computations in streaming fashion. Altogether, we prove that the robustness problem is PSPACE-complete.

1998 ACM Subject Classification D.2.4 Software/Program Verification, D.1.3 Concurrent Programming, F.4.3 Formal Languages

Keywords and phrases PGAS, SC preservation, Robustness, Semantics, Formal languages

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2013.127

1 Introduction

Partitioned global address space (PGAS) is a parallel programming model for the development of high-performance software on clusters. The PGAS model provides a global address space to the programmer that is partitioned among the cluster nodes (see Figure 1(b)). Nodes can read and write their local memories, but additionally access the remote address

* The second author was granted by the Competence Center High Performance Computing and Visualization (CC-HPC) of the Fraunhofer Institute for Industrial Mathematics (ITWM). The work was partially supported by the PROCOPE project ROIS: Robustness under Realistic Instruction Sets.

33rd Int'l Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2013).
Editors: Anil Seth and Nisheeth K. Vishnoi; pp. 127–139

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

```
int x = 1, y = 0;
write(x, rightNeighborRank,
      y, myWriteQ);
barrier();
assert(y == 1);
```

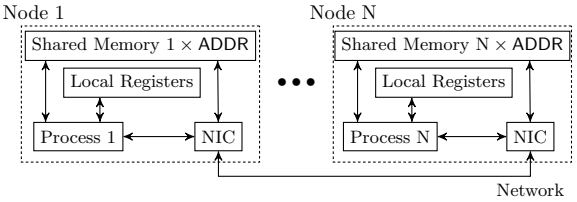

Figure 1 (a) Program **1to1** is the *compute and exchange results* idiom often found in PGAS applications. Each process copies an integer value to its neighbor. `write` asks the hardware to copy the value of address x to y on the right neighboring node. `barrier` blocks until all processes reach the barrier. The assertion can fail, as the `barrier` may return before the `write` completes. (b) PGAS architecture — NIC stands for network interface controller.

space through (synchronous or asynchronous) API calls. PGAS is a popular programming model, and supported by many PGAS APIs, such as SHMEM [10], ARMCI [21], GASNET [4], GPI [19], and GASPI [15], as well as by languages for high-performance computing, such as UPC [11], Titanium [16], and Co-Array Fortran [23].

A key ingredient of PGAS APIs is their support for one-sided communication: a node may directly read and write the memory located at a remote node without explicit synchronization with the remote side, unlike in traditional message passing interfaces. One-sided communication can be efficiently implemented on top of networking hardware featuring remote direct memory access (RDMA), and increases performance of PGAS programs by avoiding unnecessary synchronization between the sender and the receiver [19, 14].

However, the use of one-sided communication introduces additional non-determinism in the ordering of memory reads and writes, and makes reasoning about program correctness harder. Figure 1(a) demonstrates a subtle bug arising out of improper synchronizations: while the barriers ensure all processes are at the same control location, the remote writes may or may not have completed when address `y` is accessed after the barrier.

We make two contributions in this paper.

First, we provide a core calculus of PGAS APIs that models concurrent processes sharing a global address space and accessing remote memory through one-sided reads and writes. Despite the popularity of PGAS APIs in the high-performance computing community, to the best of our knowledge, there were no formal models for common PGAS APIs.

Second, we define and study a correctness criterion called *robustness* for PGAS programs. To understand robustness, we begin with a classical and intuitive correctness condition, *sequential consistency* [18]. A computation is sequentially consistent if its memory accesses happen atomically and in the order in which they were issued. Sequential consistency is too strong a criterion for PGAS programs, where time is required to access remote memory and accesses themselves can be reordered. Robustness is the weaker notion that all computations of the program have the same happens-before (data and control) dependencies [26] as some sequentially consistent computation. Our notion of robustness captures common programming error patterns [13, 20], and is derived from a similar notion in shared memory multiprocessing [26]. Related correctness criteria have been proposed for weak memory models [2, 3, 5, 6, 7, 8, 24].

A simpler correctness property would be *data race freedom* (DRF), in which no two processes access the same address at the same time, with at least one access being a write [1]. Indeed, programs free of data races are sequentially consistent. Unfortunately, DRF is too strong a requirement in practice [25], and leads to numerous false alarms. Many common synchronization idioms for PGAS programs, such as producer-consumer synchronization, and many concurrent data structure implementations, contain benign data races. Instead,

the notion of robustness captures the intuitive requirement that, even when events are reordered in a computation, there are no causality cycles. Our notion of causality is the standard *happens-before* relation from [26].

We study the algorithmic verification of robustness. Our main result is that robustness is decidable (actually PSPACE-complete) for PGAS programs, assuming a finite data domain and finite memory. Note that our model of PGAS programs is infinite-state even when the data domain is finite: one-sided communication allows unboundedly many requests to be in flight simultaneously (a feature modeled in our formalism using unbounded queues).

Our decidability result uses two technical ingredients. First, we show that among all computations violating robustness, there is always one in a certain normal form. The normal form partitions the violating computation into phases: the first phase initiates memory reads and writes, and the latter phases complete the reads and writes in the same order in which they were initiated.

Second, we provide an algorithm to detect violating computations in this normal form. We take a language-theoretic view, and introduce a multiheaded automaton model which can accept violating computations in normal form. Then the problem of checking robustness reduces to checking emptiness for multiheaded automata. Interestingly, since the normal form maintains orderings of accesses, the multiple heads can be exploited to accept violating computations without explicitly modeling unbounded queues of memory access requests. The resulting class of languages contains non-context-free ones (such as $a^n b^n c^n$), but retains sufficient decidability properties. Altogether this yields a PSPACE decision procedure for checking robustness of programs using PGAS APIs.

For lack of space, full constructions and proofs are given in [9].

Related Work. Although PGAS APIs are popular in the high-performance computing community [4, 10, 15, 19, 21], no previous work provides a unifying formal semantics that incorporates one-sided asynchronous communication. As for synchronization correctness, Park et al. proposed a testing framework for data race detection and implemented it for the UPC language [25]. However, these authors note that many data races are actually not harmful, and support the statement by the analysis of the NAS Parallel Benchmarks [22]. For this reason, in contrast to data race freedom [1], we consider robustness as a more precise notion of appropriate synchronization. Several examples from [25] show that harmful data races (like in the `knapsack` example) lead to non-robustness, while benign data races (like in the examples `NPB 3.3 BT` and `SP`) do not.

The robustness problem was posed by Shasha and Snir [26] for shared memory multiprocessing. They showed that non-sequentially consistent computations have a happens-before cycle. Alglave and Maranget [2, 3] extended this result. They developed a general theory for reasoning about robustness problems, even among different architectures. Owens [24] proposed a notion of appropriate synchronization that is based on triangular data races. Compared to robustness, triangular race freedom requires heavier synchronization, which is undesirable for performance reasons.

We consider here the algorithmic problem of checking robustness. For programs running on weak memory models the problem has been addressed in [3, 7, 8], but none of these works provides a (sound and complete) decision procedure. The first complete algorithm for checking robustness of programs running on Total Store Ordering (TSO) architectures was given in [6]. It is based on the following locality property. If a TSO program is not robust, then there is a violating computation where only one process delays commands. This insight leads to a reduction of robustness to reachability in the sequential consistency model [5]. PGAS programs allow more reorderings than TSO ones and, as a consequence, locality

does not hold. Instead, our decision procedure relies on a more complex normal form for computations and on an automata-theoretic algorithm to look for normal-form violations.

2 PGAS Programs

2.1 Features of PGAS Programs

PGAS programs are *single program, multiple data* programs running on a cluster (see Figure 1(b)). At run time, a PGAS program consists of multiple processes executing the same code on different nodes. Each process has a *rank*, which is the index of the node it runs on. The processes can access a global address space partitioned into local address spaces for each process. Local addresses can be accessed directly. Remote addresses (addresses belonging to different processes) are accessed using API calls, which come in different flavors.

SHMEM [10] provides synchronous remote reads where the invoking process waits for completion of the command. Remote write commands are asynchronous, and no ordering is guaranteed between writes, even to the same remote node. The ordering can, however, be enforced by a special fence command.

ARMCI [21] features synchronous as well as asynchronous read and write commands. The asynchronous variants of the commands return a handle that can be waited upon. When the wait on a read handle is over, the data being read has arrived and is accessible. When the wait on a write handle is over, the data being written has been sent to the network but might not have reached its destination. Unlike operations to different nodes, operations to the same remote node are executed in their issuing order.

GASNet [4], like ARMCI, provides both synchronous and asynchronous versions of reads and writes. Commands return a handle that can be waited upon, and a return from a wait implies full completion of the operation. The order in which asynchronous operations complete is intentionally left unspecified.

GPI [19] and GASPI [15] only support asynchronous read and write commands. Each read or write operation is assigned a queue identifier. In GPI, operations with the same queue id and to the same remote node are executed in the order in which they were issued; in GASPI this guarantee does not hold. One can wait on a queue id, and the wait returns when all commands in the queue are fully completed, on both the local and the remote side.

Summing up, in a uniform PGAS programming model it should be possible to

- perform synchronous and asynchronous data transfers,
- assign an asynchronous operation a handle or a queue id,
- wait for completion of an individual command or of all commands in a given queue,
- enforce ordering between operations.

We define a core model for PGAS that supports all these features. Our model only uses asynchronous remote reads and writes with explicit queues, but is flexible enough to accommodate all the above idioms. Moreover, it is not limited to single program, multiple data programs common in PGAS applications, but can model ordinary concurrent programs with different processes as well.

2.2 Syntax of PGAS Programs

We define PGAS programs and their semantics in terms of automata. A (non-deterministic) *automaton* is a tuple $A = (S, \Sigma, \Delta, s_0, F)$, where S is a set of states, Σ is a finite alphabet, $\Delta \subseteq S \times (\Sigma \cup \{\varepsilon\}) \times S$ is a set of transitions, $s_0 \in S$ is an initial state, and $F \subseteq S$ is a set of final states. We call the automaton *finite* if the set of states is finite. We write $s_1 \xrightarrow{a} s_2$

$\langle cmd \rangle ::= \langle reg \rangle \leftarrow \texttt{mem[}\langle expr \rangle\texttt{]}$
| $\texttt{mem[}\langle expr \rangle\texttt{]} \leftarrow \langle expr \rangle$
| $\langle reg \rangle \leftarrow \langle expr \rangle$
| $\texttt{assume(}\langle expr \rangle\texttt{)}$
| $\texttt{read(}\langle local\text{-}addr \rangle\texttt{,}\langle rank \rangle\texttt{,}\langle remote\text{-}addr \rangle\texttt{,}\langle que\text{-}id \rangle\texttt{)}$
| $\texttt{write(}\langle local\text{-}addr \rangle\texttt{,}\langle rank \rangle\texttt{,}\langle remote\text{-}addr \rangle\texttt{,}\langle que\text{-}id \rangle\texttt{)}$
| `barrier`

Figure 2 Syntax of commands. $\langle reg \rangle$ ranges over REG; expressions $\langle expr \rangle$, local addresses $\langle local\text{-}addr \rangle$, remote addresses $\langle remote\text{-}addr \rangle$, and queue identifiers $\langle que\text{-}id \rangle$ range over expressions; ranks $\langle rank \rangle$ over $\overline{1,N}$-valued expressions.

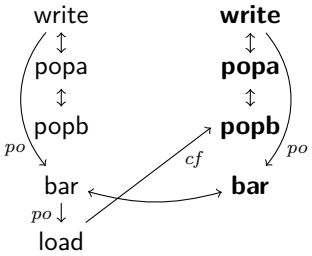

Figure 3 Happens-before relation of computation $\tau_{\texttt{1to1}}$ (Example 1). $\tau_{\texttt{1to1}}$ violates robustness.

if $(s_1, a, s_2) \in \Delta$, and extend the relation to computations $\sigma \in \Sigma^*$ in the expected way. The *language* of the automaton is $\mathcal{L}(A) := \{\sigma \in \Sigma^* \mid s_0 \xrightarrow{\sigma} s \text{ for some } s \in F\}$. We write $|\sigma|$ for the length of a computation $\sigma \in \Sigma^*$, and use $\mathsf{succ}(\sigma)$ to denote the successor relation among the letters in σ. We write $a <_\sigma b$ if $\sigma = \sigma_1 \cdot a \cdot \sigma_2 \cdot b \cdot \sigma_3$ for some $\sigma_1, \sigma_2, \sigma_3 \in \Sigma^*$.

A *PGAS program* $(\mathcal{P}, N)$ consists of a program code $\mathcal{P}$ and a fixed number $N \geq 1$ of cluster nodes. The *program code* $\mathcal{P} := (Q, \mathsf{CMD}, \mathcal{I}, q_0, Q)$ is a finite automaton with a set of control states Q, all of them final, an initial state q_0, and a set of transitions $\mathcal{I}$ labeled with *commands* CMD defined as follows.

Let DOM, ADDR, and QUE be finite domains of values (containing a value 0), addresses, and queue identifiers, respectively. Let REG be a finite set registers that take values from DOM. The grammar of commands is given in Figure 2. For simplicity, we will assume $\mathsf{DOM} = \mathsf{ADDR} = \mathsf{QUE}$. The set of expressions is defined over constants from DOM, registers from REG, and (unspecified) operators over DOM. The set of commands CMD includes local assignments and conditionals (`assume`), remote read and write API calls `read` and `write` respectively, and barriers `barrier`.

At run time, there is a process on each node $\overline{1,N}$ that executes program $\mathcal{P}$, where $\overline{M,N} := \{M, M+1, \ldots, N\}$. We will identify each process with its rank from $\mathsf{RNK} := \overline{1,N}$. For modeling purposes, one may assume there are special constant expressions that let a process learn about its rank in RNK and about the total number of processes N.

2.3 Semantics of PGAS Programs

The semantics of a PGAS program $(\mathcal{P}, N)$ is defined using a *state-space automaton* $X(\mathcal{P}, N) := (S_X, \mathsf{E}, \Delta_X, s_{0X}, F_X)$. A state $s \in S_X$ is a tuple $s = (\mathsf{st}, \mathsf{m}, \mathsf{fa}, \mathsf{fb})$, where state configuration $\mathsf{st}\colon \mathsf{RNK} \to Q$ maps each process to its current control state, memory configuration $\mathsf{m}\colon \mathsf{RNK} \times (\mathsf{REG} \cup \mathsf{ADDR}) \to \mathsf{DOM}$ maps each process to the values stored in each register and at each address, queue configuration $\mathsf{fa}\colon \mathsf{RNK} \times \mathsf{QUE} \to (\mathsf{RNK} \times \mathsf{ADDR} \times \mathsf{RNK} \times \mathsf{ADDR})^*$ maps each process to remote read and write requests that were issued, and $\mathsf{fb}\colon \mathsf{RNK} \times \mathsf{QUE} \to (\mathsf{RNK} \times \mathsf{ADDR} \times \mathsf{DOM})^*$ contains values to be transferred. The two queue configurations capture the delays between creating a request, reading data, and writing data.

The initial state is $s_{0X} := (\mathsf{st}_0, \mathsf{m}_0, \mathsf{fa}_0, \mathsf{fb}_0)$, where for all ranks $\mathsf{r} \in \mathsf{RNK}$, registers and addresses $\mathsf{a} \in \mathsf{REG} \cup \mathsf{ADDR}$, and queue identifiers $\mathsf{q} \in \mathsf{QUE}$, we have $\mathsf{st}_0(\mathsf{r}) := q_0$, $\mathsf{m}_0(\mathsf{r}, \mathsf{a}) := 0$, and $\mathsf{fa}_0(\mathsf{r}, \mathsf{q}) := \varepsilon =: \mathsf{fb}_0(\mathsf{r}, \mathsf{q})$. The set of final states is $F_X := \{(\mathsf{st}, \mathsf{m}, \mathsf{fa}, \mathsf{fb}) \in S_X \mid \mathsf{fa}(\mathsf{r}, \mathsf{q}) = \varepsilon = \mathsf{fb}(\mathsf{r}, \mathsf{q}) \text{ for all } \mathsf{r} \in \mathsf{RNK}, \mathsf{q} \in \mathsf{QUE}\}$. The semantics of commands ensures queues can always be emptied, so acceptance with empty queues is not a restriction.

The alphabet of $X(\mathcal{P}, N)$ is the set of *events* $\mathsf{E} := \mathsf{K} \times \mathsf{RNK} \times ((\mathsf{RNK} \times \mathsf{ADDR}) \cup \{\bot\})$ with *event kinds* $\mathsf{K} := \{\mathsf{load}, \mathsf{store}, \mathsf{assign}, \mathsf{assume}, \mathsf{read}, \mathsf{write}, \mathsf{popa}, \mathsf{popb}, \mathsf{bar}\}$. Consider an event

Table 1 Transition rules for $X(\mathcal{P}, N)$, given $q_1 \xrightarrow{\mathsf{cmd}} q_2$ and current state $s = (\mathsf{st}, \mathsf{m}, \mathsf{fa}, \mathsf{fb})$ with $\mathsf{st}(\mathsf{r}) = q_1$. We set $\mathsf{st}' := \mathsf{st}[\mathsf{r} := q_2]$ to update st so that process r is at q_2. $\widehat{e}$ denotes the evaluation of expression e in memory configuration m.

$$\frac{\mathsf{cmd} = r \leftarrow \mathtt{mem}[e_\mathsf{a}]}{s \xrightarrow{(\mathsf{load},\mathsf{r},(\mathsf{r},\widehat{e_\mathsf{a}}))} (\mathsf{st}', \mathsf{m}[(\mathsf{r}, r) := \mathsf{m}(\mathsf{r}, \widehat{e_\mathsf{a}})], \mathsf{fa}, \mathsf{fb})} \quad (\mathsf{load})$$

$$\frac{\mathsf{cmd} = \mathtt{write}(e_\mathsf{a}^{\mathsf{loc}}, e_\mathsf{r}^{\mathsf{rem}}, e_\mathsf{a}^{\mathsf{rem}}, e_\mathsf{q}) \qquad \mathsf{fa}(\mathsf{r}, \widehat{e_\mathsf{q}}) = \alpha}{s \xrightarrow{(\mathsf{write},\mathsf{r},\bot)} (\mathsf{st}', \mathsf{m}, \mathsf{fa}[(\mathsf{r}, \widehat{e_\mathsf{q}}) := \alpha \cdot (\mathsf{r}, \widehat{e_\mathsf{a}^{\mathsf{loc}}}, \widehat{e_\mathsf{r}^{\mathsf{rem}}}, \widehat{e_\mathsf{a}^{\mathsf{rem}}})], \mathsf{fb})} \quad (\mathsf{write})$$

$$\frac{\mathsf{fa}(\mathsf{r},\mathsf{q}) = (\mathsf{r}_\mathsf{s}, \mathsf{a}_\mathsf{s}, \mathsf{r}_\mathsf{d}, \mathsf{a}_\mathsf{d}) \cdot \alpha \qquad \mathsf{fb}(\mathsf{r},\mathsf{q}) = \beta}{s \xrightarrow{(\mathsf{popa},\mathsf{r},(\mathsf{r}_\mathsf{s},\mathsf{a}_\mathsf{s}))} (\mathsf{st}, \mathsf{m}, \mathsf{fa}[(\mathsf{r},\mathsf{q}) := \alpha], \mathsf{fb}[(\mathsf{r},\mathsf{q}) := \beta \cdot (\mathsf{r}_\mathsf{d}, \mathsf{a}_\mathsf{d}, \mathsf{m}(\mathsf{r}_\mathsf{s}, \mathsf{a}_\mathsf{s}))])} \quad (\mathsf{popa})$$

$$\frac{\mathsf{fb}(\mathsf{r},\mathsf{q}) = (\mathsf{r}_\mathsf{d}, \mathsf{a}_\mathsf{d}, v) \cdot \beta}{s \xrightarrow{(\mathsf{popb},\mathsf{r},(\mathsf{r}_\mathsf{d},\mathsf{a}_\mathsf{d}))} (\mathsf{st}, \mathsf{m}[(\mathsf{r}_\mathsf{d}, \mathsf{a}_\mathsf{d}) := v], \mathsf{fa}, \mathsf{fb}[(\mathsf{r},\mathsf{q}) := \beta])} \quad (\mathsf{popb})$$

$$\frac{\mathsf{st}(\mathsf{r}) \xrightarrow{\mathtt{barrier}} \mathsf{st}'(\mathsf{r}) \text{ for each } \mathsf{r} \in \mathsf{RNK}}{s \xrightarrow{(\mathsf{bar},1,\bot)\cdot(\mathsf{bar},2,\bot)\cdots(\mathsf{bar},N,\bot)} (\mathsf{st}', \mathsf{m}, \mathsf{fa}, \mathsf{fb})} \quad (\mathsf{bar})$$

$\mathsf{e} = (\mathsf{k}, \mathsf{r}, (\mathsf{r}_\mathsf{a}, \mathsf{a})) \in \mathsf{E}$. We use $\mathsf{kind}(\mathsf{e}) = \mathsf{k}$ to determine the kind of the event, $\mathsf{rank}(\mathsf{e}) = \mathsf{r}$ for the rank of the process that produced the event, and $\mathsf{addr}(\mathsf{e}) = (\mathsf{r}_\mathsf{a}, \mathsf{a})$ to obtain the rank and the address that are *accessed* by the event. If $\mathsf{kind}(\mathsf{e}) \in \{\mathsf{load}, \mathsf{popa}\}$, then e is said to be a *read of* (r_a, a). If $\mathsf{kind}(\mathsf{e}) \in \{\mathsf{store}, \mathsf{popb}\}$, then e is a *write of* address $\mathsf{addr}(\mathsf{e})$.

Table 1 shows a subset of the transition relation Δ_X; the remaining rules are similar. When a process executes a remote write command, Rule (write), a new item is added to a queue in fa. This item contains the source rank and source address from which the data will be copied, together with the destination rank and destination address to which the data will be copied. Eventually, the item is popped from the queue in fa, Rule (popa), the value is read from the source address, and a new item is pushed into the corresponding queue in fb. The new item contains the destination rank and destination address, and the value that was read from the source address. Eventually, this item is popped from the queue, Rule (popb), and the value is written to the destination address in the destination rank. Modeling two queue configurations yields a symmetry between remote writes and reads: a read can be interpreted as a write that comes upon request.

The semantics of a PGAS program $\mathsf{C}(\mathcal{P}, N) := \mathcal{L}(X(\mathcal{P}, N)) \subseteq \mathsf{E}^*$ is the set of computations of the state-space automaton.

▶ **Example 1.** Consider PGAS program $(\mathbf{1to1}, 2)$ with the program code from Figure 1(a) being run on two nodes. It has the following computation:

$$\tau_{\mathbf{1to1}} = \mathsf{write} \cdot \mathbf{write} \cdot \mathbf{popa} \cdot \mathsf{popa} \cdot \mathsf{bar} \cdot \mathbf{bar} \cdot \mathsf{load} \cdot \mathbf{popb} \cdot \mathsf{popb}.$$

Bold events belong to the process with rank 2, the other events — to the process with rank 1. We have $\mathsf{addr}(\mathsf{popa}) = (1, x)$, $\mathsf{addr}(\mathsf{popb}) = (2, y)$. Symmetrically, $\mathsf{addr}(\mathbf{popa}) = (2, x)$ and $\mathsf{addr}(\mathbf{popb}) = (1, y)$. The `assert` in Figure 1 is a shortcut for a combination of load and assume, and in this computation $\mathsf{addr}(\mathsf{load}) = (1, y)$.

2.4 Simulating PGAS APIs

Our formalism natively supports asynchronous data transfers and queues. Operations in the same queue are completed in the order in which they were issued. Using this, we can model

the ordering guarantees given by ARMCI and GPI – by putting ordered operations into the same queue.

To model waiting on individual operations (waiting on a handle), we associate a shadow memory address with each operation. Before issuing the operation, the value at this address is set to 0. When the operation has been issued, the process sends to the same queue a read request which overwrites the value at the shadow address to 1. Now waiting on the individual operation can be implemented by polling on the shadow address associated with the operation. Waiting on all operations in a given queue is done similarly. Synchronous data transfers are modeled by asynchronous transfers, immediately followed by a wait.

3 Robustness: A Notion of Appropriate Synchronization

We now define *robustness*, a correctness condition for PGAS programs. Robustness is a weaker criterion than requiring all computations to be sequentially consistent [18]: it allows for reordering of events as long as there are no causality cycles. As causality relation, we adopt the *happens-before relation* [26]. Fix a computation $\tau \in \mathsf{C}(\mathcal{P}, N)$. Its happens-before relation is the union of the three relations we define next, $\rightarrow_{hb}(\tau) := \rightarrow_{po} \cup \rightarrow_{cf} \cup \leftrightarrow$.

The *program order relation* $\rightarrow_{po}$ is the union of the program order relations for all processes: $\rightarrow_{po} := \bigcup_{\mathsf{r} \in \mathsf{RNK}} \rightarrow^{\mathsf{r}}_{po}$. Relation $\rightarrow^{\mathsf{r}}_{po}$ gives the order in which events were issued in process r. Formally, let τ' be the subsequence of all events e in τ such that $\mathsf{rank}(\mathsf{e}) = \mathsf{r}$ and $\mathsf{kind}(\mathsf{e}) \notin \{\mathsf{popa}, \mathsf{popb}\}$. Then $\rightarrow^{\mathsf{r}}_{po} := \mathsf{succ}(\tau')$.

The *conflict relation* $\rightarrow_{cf}$ orders conflicting accesses to the same address. Let $\tau = \alpha \cdot \mathsf{e}_1 \cdot \beta \cdot \mathsf{e}_2 \cdot \gamma$, where e_1 and e_2 access the same address, and at least one of them is a write: $\mathsf{addr}(\mathsf{e}_1) = \mathsf{addr}(\mathsf{e}_2) = (\mathsf{r}, \mathsf{a})$, $\mathsf{kind}(\mathsf{e}_1) \in \{\mathsf{store}, \mathsf{popb}\}$ or $\mathsf{kind}(\mathsf{e}_2) \in \{\mathsf{store}, \mathsf{popb}\}$. If there is no $\mathsf{e} \in \beta$ such that $\mathsf{addr}(\mathsf{e}) = (\mathsf{r}, \mathsf{a})$ and $\mathsf{kind}(\mathsf{e}) \in \{\mathsf{store}, \mathsf{popb}\}$, then $\mathsf{e}_1 \rightarrow_{cf} \mathsf{e}_2$.

The *identity relation* $\leftrightarrow$ identifies events corresponding to the same command. Let e be a remote read or write event, $\mathsf{kind}(\mathsf{e}) \in \{\mathsf{read}, \mathsf{write}\}$, and e_1 and e_2 be the corresponding requests, $\mathsf{kind}(\mathsf{e}_1) = \mathsf{popa}$ and $\mathsf{kind}(\mathsf{e}_2) = \mathsf{popb}$. Then we have $\mathsf{e} \leftrightarrow \mathsf{e}_1 \leftrightarrow \mathsf{e}_2$. In a similar way, $\leftrightarrow$ identifies matching barrier events in different processes.

We say a computation τ is *violating* if the associated happens-before relation contains a non-trivial cycle, i.e., a cycle that is not included in $\leftrightarrow$. Violating computations violate sequential consistency. The robustness problem amounts to proving the absence of violations:

ROB Given a program $(\mathcal{P}, N)$, show that no computation $\tau \in \mathsf{C}(\mathcal{P}, N)$ is violating.

▶ **Example 2.** The happens-before relation of computation $\tau_{\mathbf{1to1}}$ is depicted in Figure 3. It is cyclic, therefore $\tau_{\mathbf{1to1}}$ is violating and $(\mathbf{1to1}, 2)$ is not robust. Indeed, no sequentially consistent execution of **1to1** allows the `assert` statements to load the initial value of y.

Our main result is the following.

▶ **Theorem 3.** **ROB** *is* PSPACE*-complete.*

The PSPACE lower bound follows from PSPACE-hardness of control state reachability in sequentially consistent programs [17]. To reduce to robustness, we add an artificial happens-before cycle starting in the target control state. The rest of the paper shows a PSPACE algorithm, and hence upper bound, for the problem.

4 Normal-Form Violations

We show that a PGAS program is not robust if and only if it has a violating computation of the following normal form.

▶ **Definition 4.** Computation $\tau = \tau_1 \cdot \tau_2 \cdot \tau_3 \cdot \tau_4 \in \mathsf{C}(\mathcal{P}, N)$ is *in normal form* if all $\mathsf{e} \in \tau_2 \cdot \tau_3 \cdot \tau_4$ satisfy $\mathsf{kind}(\mathsf{e}) \in \{\mathsf{popa}, \mathsf{popb}\}$ and for all $a, b \in \tau_1$ with $\mathsf{kind}(a), \mathsf{kind}(b) \notin \{\mathsf{popa}, \mathsf{popb}\}$ and all $a', b' \in \tau_i$ with $i \in \overline{1,4}$ we have:

$$a <_{\tau_1} b,\ a \not\leftrightarrow^* b,\ a \leftrightarrow^* a',\ b \leftrightarrow^* b' \quad \text{implies} \quad a' <_{\tau_i} b'. \tag{NF}$$

We explain the normal-form requirement (NF). Consider two accesses a and b to remote processes that can be found in the first part of the computation τ_1. Assume corresponding pop events a' and b' are delayed and can both be found in a later part of the computation, say τ_2. Then the ordering of a' and b' in τ_2 coincides with the order of a and b in τ_1. Computation $\tau_{\mathbf{1to1}}$ is not in normal-form whereas $\tau^{nf}_{\mathbf{1to1}}$ in Figure 4 is. The following theorem guarantees that, in case of non-robustness, normal-form violations always exist.

▶ **Theorem 5.** *A PGAS program $(\mathcal{P}, N)$ is robust iff it has no normal-form violation.*

Phrased differently, to decide robustness our procedure should look for normal-form violations. The remainder of the section is devoted to proving Theorem 5. We make use of the following property of PGAS programs: every computation contains an event that can be deleted, in the sense that the result is again a computation of the program, i.e., in $\mathsf{C}(\mathcal{P}, N)$.

▶ **Lemma 6** (Cancellation)**.** *Consider a computation $\varepsilon \neq \tau \in \mathsf{C}(\mathcal{P}, N)$ and let e be the last event in τ with $\mathsf{kind}(\mathsf{e}) \notin \{\mathsf{popa}, \mathsf{popb}\}$. Then $\tau \setminus \mathsf{e} \in \mathsf{C}(\mathcal{P}, N)$, where computation $\tau \setminus \mathsf{e}$ is defined to remove e and all $\leftrightarrow$-related events from τ.*

Proof. All events to the right of e are unconditionally executable. Moreover, τ does not have $\rightarrow_{po}$-successors following e. Therefore, the resulting computation $\tau \setminus \mathsf{e}$ is in $\mathsf{C}(\mathcal{P}, N)$. ◀

A PGAS program is not robust if and only if it has a violating computation τ of minimal length. Let $\mathsf{e} \in \tau$ be the event determined by Lemma 6. If $\mathsf{kind}(\mathsf{e}) \notin \{\mathsf{read}, \mathsf{write}\}$, then $\tau = \tau_1 \cdot \mathsf{e} \cdot \tau_2$. Otherwise $\tau = \tau_1 \cdot \mathsf{e} \cdot \tau_2 \cdot \mathsf{e}' \cdot \tau_3 \cdot \mathsf{e}'' \cdot \tau_4$ with $\mathsf{e} \leftrightarrow \mathsf{e}' \leftrightarrow \mathsf{e}''$. Consider the latter case where $\tau \setminus \mathsf{e} = \tau_1 \cdot \tau_2 \cdot \tau_3 \cdot \tau_4$. Since $|\tau \setminus \mathsf{e}| < |\tau|$, the new computation is not violating and $\rightarrow_{hb} (\tau \setminus \mathsf{e})$ is acyclic. This acyclicity guarantees that we find a computation $\sigma \in \mathsf{E}^*$ with the same happens-before relation as $\tau \setminus \mathsf{e}$ and where pop events directly follow their remote accesses. Intuitively, σ is a sequentially consistent computation corresponding to $\tau \setminus \mathsf{e}$.

▶ **Lemma 7** ([26])**.** *There is $\sigma \in \mathsf{C}(\mathcal{P}, N)$ with $\rightarrow_{hb} (\sigma) = \rightarrow_{hb} (\tau \setminus \mathsf{e})$ and $\sigma = \sigma_1 \cdot \mathsf{e}_1 \ldots \mathsf{e}_n \cdot \sigma_2$ for all $\mathsf{e}_1 \leftrightarrow \ldots \leftrightarrow \mathsf{e}_n$.*

We now use σ to rearrange the events in $\tau \setminus \mathsf{e}$ and guarantee the normal-form requirement. The idea is to project σ to the events in τ_1 to τ_4. Reinserting e yields a normal-form violation:

$$\tau^{nf} := (\sigma \downarrow \tau_1) \cdot \mathsf{e} \cdot (\sigma \downarrow \tau_2) \cdot \mathsf{e}' \cdot (\sigma \downarrow \tau_3) \cdot \mathsf{e}'' \cdot (\sigma \downarrow \tau_4).$$

The following lemma concludes the proof of Theorem 5.

▶ **Lemma 8** (Reinsertion)**.** *$\tau^{nf} \in \mathsf{C}(\mathcal{P}, N)$, $\rightarrow_{hb} (\tau^{nf}) = \rightarrow_{hb} (\tau)$, and τ^{nf} is in normal form.*

▶ **Example 9.** Computation $\tau_{\mathbf{1to1}}$ in Example 1 is a shortest violation. The event determined by Lemma 6 is $\mathsf{e} = \mathsf{load}$. Therefore, $\tau \setminus \mathsf{e} = \tau_1 \cdot \tau_2$ with

$$\tau_1 = \mathsf{write} \cdot \mathbf{write} \cdot \mathbf{popa} \cdot \mathsf{popa} \cdot \mathsf{bar} \cdot \mathbf{bar} \qquad \text{and} \qquad \tau_2 = \mathbf{popb} \cdot \mathsf{popb}.$$

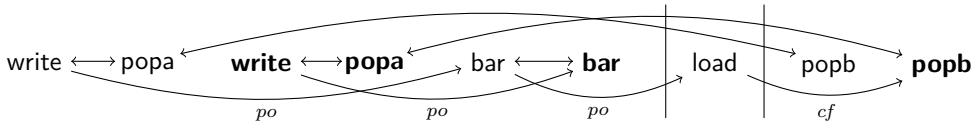

Figure 4 Normal-form violation $\tau^{nf}_{\mathbf{1to1}}$ from Example 9. The edges indicate the dependencies in the computation and coincide with the relations in Figure 3.

A sequentially consistent computation corresponding to $\tau \setminus \mathsf{e}$ is

$$\sigma = \mathsf{write} \cdot \mathsf{popa} \cdot \mathsf{popb} \cdot \mathbf{write} \cdot \mathbf{popa} \cdot \mathbf{popb} \cdot \mathsf{bar} \cdot \mathbf{bar}.$$

The normal-form violation $\tau^{nf}_{\mathbf{1to1}}$ is depicted in Figure 4. Note that $\tau^{nf}_{\mathbf{1to1}}$ is indeed in $\mathsf{C}(\mathbf{1to1}, 2)$. Moreover, popa and **popa** immediately follow write and **write**, respectively. Similarly, the popb and **popb** events in the second part of the computation respect the order of write and **write** in the first part of the computation. This means the property (NF) holds.

5 From Normal-Form Violations to Language Emptiness

We now reduce checking the absence of normal-form violations to the emptiness problem in a suitable automaton model. We introduce multiheaded automata and construct, for each program $(\mathcal{P}, N)$, a multiheaded automaton accepting all normal-form computations. To verify robustness, we check that the intersection of this automaton with regular languages accepting cyclic happens-before relations is empty.

5.1 Multiheaded Automata

Multiheaded automata are an extension of finite automata. Intuitively, instead of generating just a single computation, they generate several computations in one pass, each by a separate head. The language of the multiheaded automaton then consists of the concatenations of the computations generated by each head.

Syntactically, an *n-headed finite automaton over* Σ is a finite automaton that uses the extended alphabet $\overline{1,n}\times\Sigma$. So we have $A = (S, (\overline{1,n}\times\Sigma), \Delta, s_0, F)$. The semantics, however, is different from finite automata. Given $\sigma \in (\overline{1,n}\times\Sigma)^*$, we use $\sigma{\downarrow}k$ to project σ to the letters (k, a), and afterwards cut away the index k. So $((1,a)\cdot(2,b)\cdot(1,c)){\downarrow}1 = a\cdot c$. The *language of* A is $\mathcal{L}(A) := \{\mathsf{comp}(\sigma) \mid s_0 \xrightarrow{\sigma} s \text{ for some } s \in F\}$ where $\mathsf{comp}(\sigma) := \sigma{\downarrow}1 \cdots \sigma{\downarrow}n$.

Multiheaded automata are closed under regular intersection, and emptiness is decidable in non-deterministic logarithmic space. Indeed, checking emptiness reduces to finding a path from an initial to a final node in a directed graph.

▶ **Lemma 10.** *Consider an n-headed automaton U and a finite automaton V over a common alphabet Σ. There is an n-headed automaton W with $\mathcal{L}(W) = \mathcal{L}(U) \cap \mathcal{L}(V)$.*

▶ **Lemma 11.** *Emptiness for n-headed automata is* NL*-complete.*

Multiheaded automata are incomparable with context-free grammars, and indeed the normal-form computations of a program may be non-context-free.[1]

[1] Consider $\mathcal{P} := (\{q_0\}, \mathsf{CMD}, \{q_0 \xrightarrow{\mathtt{read}(0,0,0,0)} q_0\}, \{q_0\})$ running on a single node. The language $\mathsf{C}(\mathcal{P}, 1)$ is not context-free. To see this, let $\mathsf{kind}(a) = \mathsf{read}$, $\mathsf{kind}(b) = \mathsf{popa}$, and $\mathsf{kind}(c) = \mathsf{popb}$. Then $\mathsf{C}(\mathcal{P}, 1) \cap a^*b^*c^*$ is the non-context-free language $\{a^p b^p c^p \mid p \geq 0\}$.

Table 2 Transition rules for $Y(\mathcal{P}, N)$, given $q_1 \xrightarrow{\mathsf{cmd}} q_2$ and current state $s = (\mathsf{st}, \mathsf{m}, \mathsf{pa}, \mathsf{pb})$ with $\mathsf{st}(\mathsf{r}) = q_1$. The target is $s' = (\mathsf{st}', \mathsf{m}', \mathsf{pa}', \mathsf{pb}')$ where, unless otherwise stated, $\mathsf{st}' = \mathsf{st}$, $\mathsf{m}' = \mathsf{m}$, $\mathsf{pa}' = \mathsf{pa}$, $\mathsf{pb}' = \mathsf{pb}$. The auxiliary states $s_{aux1}, s_{aux2} \in S_Y^{\mathsf{aux}}$ are unique for each rule application.

$$(\mathsf{gpa}') \ \frac{\mathsf{pa}(\mathsf{r},\mathsf{q}) < \mathsf{pb}(\mathsf{r},\mathsf{q})}{s \xrightarrow{\varepsilon} s' \ \mathsf{pa}' := \mathsf{pa}[(\mathsf{r},\mathsf{q}) := \mathsf{pa}(\mathsf{r},\mathsf{q}) + 1]} \qquad \frac{\mathsf{pb}(\mathsf{r},\mathsf{q}) < 4}{s \xrightarrow{\varepsilon} s' \ \mathsf{pb}' := \mathsf{pb}[(\mathsf{r},\mathsf{q}) := \mathsf{pb}(\mathsf{r},\mathsf{q}) + 1]} \ (\mathsf{gpb}')$$

$$\frac{\mathsf{cmd} = \mathtt{write}(e_{\mathsf{a}}^{\mathsf{loc}}, e_{\mathsf{r}}^{\mathsf{rem}}, e_{\mathsf{a}}^{\mathsf{rem}}, e_{\mathsf{q}}) \qquad \mathsf{pa}(\mathsf{r}, \widehat{e_{\mathsf{q}}}) = m \qquad \mathsf{pb}(\mathsf{r}, \widehat{e_{\mathsf{q}}}) = n}{s \xrightarrow{1,(\mathsf{write},\mathsf{r},\perp)} s_{aux1} \xrightarrow{m,(\mathsf{popa},\mathsf{r},(\mathsf{r},\widehat{e_{\mathsf{a}}^{\mathsf{loc}}}))} s_{aux2} \xrightarrow{n,(\mathsf{popb},\mathsf{r},(\widehat{e_{\mathsf{r}}^{\mathsf{rem}}},\widehat{e_{\mathsf{a}}^{\mathsf{rem}}}))} s' \quad \mathsf{st}' := \mathsf{st}[\mathsf{r} := q_2]} \ (\mathsf{write}')$$
$$\text{if } n = 1 \text{ then } \mathsf{m}' := \mathsf{m}[(\widehat{e_{\mathsf{r}}^{\mathsf{rem}}}, \widehat{e_{\mathsf{a}}^{\mathsf{rem}}}) := \mathsf{m}(\mathsf{r}, \widehat{e_{\mathsf{a}}^{\mathsf{loc}}})]$$

Multiheaded automata can be understood as a restriction of matrix grammars [12]. In matrix grammars, productions simultaneously rewrite multiple non-terminals. Roughly, each production can be understood as a Petri net transition, and emptiness is decidable as Petri net reachability is. Since we target a PSPACE result, matrix grammars are too expressive for our purposes.

5.2 Detecting Normal-Form Computations

We define a 4-headed automaton $Y(\mathcal{P}, N) := (S_Y \uplus S_Y^{\mathsf{aux}}, \mathsf{E}, \Delta_Y, s_{0Y}, S_Y)$ that accepts all normal-form computations $\tau = \tau_1 \cdot \tau_2 \cdot \tau_3 \cdot \tau_4 \in \mathsf{C}(\mathcal{P}, N)$. In order to accept τ_1, the new automaton tracks the control and memory configurations in the way $X(\mathcal{P}, N)$ does. For the remainder of the computation, these configurations are not needed. Indeed, τ_2 to τ_4 only consist of popa and popb events that are executable independently of the control and memory configurations. However, $Y(\mathcal{P}, N)$ has to take care of the ordering of popa and popb events from the same queue. In particular, if e_1 handles a request issued before the request of e_2 with $\mathsf{kind}(\mathsf{e}_1) = \mathsf{kind}(\mathsf{e}_2)$, then it cannot be the case that $\mathsf{e}_1 \in \tau_j$ and $\mathsf{e}_2 \in \tau_i$ with $i < j$.

Guided by this discussion, we define a state $s \in S_Y$ as a tuple $s := (\mathsf{st}, \mathsf{m}, \mathsf{pa}, \mathsf{pb})$. The state and memory configurations st and m are defined as in Section 2. They reflect the state of the program after it has generated a prefix of τ_1. The functions $\mathsf{pa}, \mathsf{pb} : \mathsf{RNK} \times \mathsf{QUE} \to \overline{1,4}$ give, for each process and each queue, the part τ_1 to τ_4 of the computation where the next popa resp. popb event will be generated. The initial state is $s_0 := (\mathsf{st}_0, \mathsf{m}_0, \mathsf{pa}_0, \mathsf{pb}_0)$ with $\mathsf{pa}_0(\mathsf{r},\mathsf{q}) := 1 =: \mathsf{pb}_0(\mathsf{r},\mathsf{q})$ for all $\mathsf{r} \in \mathsf{RNK}$ and $\mathsf{q} \in \mathsf{QUE}$.

The transition relation Δ_Y is the smallest relation defined by the rules in Table 2. Rule (gpa') lets the automaton choose the part of the computation to which the next popa event will be appended. The first restriction is that the index of the part can only increase, as events from the same queue are processed in order. The second restriction is that popa events cannot be generated to the right of popb events from the same queue. Rule (gpb') is the similar rule for popb events.

By Rule (write'), the automaton appends a write event to τ_1 and the corresponding popa and popb events in one shot to the parts determined by pa and pb. Since a single transition of a multiheaded automaton can generate at most one letter, the rule makes use of intermediary states from S_Y^{aux}. If popb is added to τ_1, the memory configuration is updated accordingly. Note that the generation in one shot causes pop events within the same part τ_i to follow in the order of the corresponding read/write events in τ_1. Fortunately, this is always the case in normal-form computations by (NF). Computations that are not in normal form, e.g. $\tau_{\mathbf{1to1}}$, cannot be generated by $Y(\mathcal{P}, N)$.

The set of final states of $Y(\mathcal{P}, N)$ is S_Y. The auxiliary states S_Y^{aux} are not included in the set of final states to forbid computations with pending remote requests.

▶ **Lemma 12.** $\{\tau \in C(\mathcal{P}, N) \mid \tau \text{ is in normal form}\} = \mathcal{L}(Y(\mathcal{P}, N))$.

5.3 Detecting Violations

The multiheaded automaton accepts all normal form computations, and we would like to check if one of these computations is violating. In general, violating computations can contain complicated cycles in the happens-before relation. However, we now show that whenever a computation has a happens-before cycle, it has a cycle in which each process is entered and left at most once. Our algorithm for robustness will look for happens-before cycles of this special form that, as we will show, can be captured by a regular language.

▶ **Lemma 13.** *Computation $\tau \in C(\mathcal{P}, N)$ is violating iff there is a cycle*

$$a_1 \leftrightarrow^* b_1 \rightarrow_{po}^* c_1 \leftrightarrow^* d_1 \rightsquigarrow \ldots \rightsquigarrow a_k \leftrightarrow^* b_k \rightarrow_{po}^* c_k \leftrightarrow^* d_k \rightsquigarrow a_1 \qquad \text{(CYC)}$$

where $\mathit{rank}(x_i) = \mathit{rank}(y_j)$ iff $i = j$, for all $x_i, y_j \in \{a_1, \ldots, d_k\}$, and $\rightsquigarrow := \rightarrow_{cf} \cup \leftrightarrow$.

▶ **Example 14.** The computations $\tau_{\mathbf{1to1}}$ (Example 1) and $\tau^{nf}_{\mathbf{1to1}}$ (Example 9) have a cycle of the form (CYC) depicted in Figure 3: $k = 2$, $a_1 = b_1 = \mathsf{bar}$, $c_1 = d_1 = \mathsf{load}$, $a_2 = \mathbf{popb}$, $b_2 = \mathbf{write}$, $c_2 = d_2 = \mathbf{bar}$.

Note that $d_i \leftrightarrow a_{i+1}$ means both are barriers, $\mathsf{kind}(d_i) = \mathsf{bar} = \mathsf{kind}(a_{i+1})$. This holds as the ranks are different. In spite of the additional restrictions, cycles (CYC) are not trivial to recognize. The reason is that the events constituting the cycle are not necessarily contained in the computation in the order in which they appear in the cycle, see Figure 4. The idea of our cycle detection is to first guess the events a_i and d_i for each process and then check that $d_i \rightsquigarrow a_{i+1}$ holds. The former can be accomplished by an extension $Y^{\mathsf{M}}(\mathcal{P}, N)$ of the multiheaded automaton $Y(\mathcal{P}, N)$, the latter by a regular intersection.

The automaton $Y^{\mathsf{M}}(\mathcal{P}, N)$ accepts computations over the alphabet $\mathsf{E} \times \mathsf{M}$ with $\mathsf{M} := 2^{\{\mathsf{enter},\mathsf{leave}\}}$. The events marked by enter are the guessed a_i events in (CYC) and those marked by leave are the d_i events in (CYC). We still have to guarantee we only mark a_i and d_i that satisfy $a_i \leftrightarrow^* b_i \rightarrow_{po}^* c_i \leftrightarrow^* d_i$. This is straightforward thanks to the fact that $Y(\mathcal{P}, N)$ generates the events of each process in program order, and generates events related by $\leftrightarrow$ in one shot. The full construction of $Y^{\mathsf{M}}(\mathcal{P}, N)$ is given in [9].

▶ **Example 15.** Consider the normal-form computation $\tau^{nf}_{\mathbf{1to1}}$ (Example 9) that has the cycle (CYC) given in Figure 3. A corresponding marked computation of $Y^{\mathsf{M}}(\mathcal{P}, N)$ is

$$(\mathsf{write}, \emptyset) \cdot (\mathsf{popa}, \emptyset) \cdot (\mathbf{write}, \emptyset) \cdot (\mathbf{popa}, \emptyset) \cdot (\mathsf{bar}, \{\mathsf{enter}\}) \cdot (\mathbf{bar}, \{\mathsf{leave}\}) \cdot (\mathsf{load}, \{\mathsf{leave}\}) \cdot (\mathsf{popb}, \emptyset) \cdot (\mathbf{popb}, \{\mathsf{enter}\}).$$

Every cycle of the form (CYC) has a *cycle type* cyc, which is a sequence $\mathsf{cyc} = \mathsf{r}_1 \ldots \mathsf{r}_k$ of ranks from $\overline{1, N}$ with $\mathsf{r}_i \neq \mathsf{r}_j$ for $i \neq j$. The idea is that the events a_i, b_i, c_i, d_i belong to rank r_i. For each pair $\mathsf{r}_i, \mathsf{r}_{i+1}$ in this sequence, we construct a finite automaton $Z^{\mathsf{r}_i,\mathsf{r}_{i+1}}$ over the alphabet $\mathsf{E} \times \mathsf{M}$. It checks whether there is a conflict or identity edge from the leave-marked event of process r_i to the enter-marked event of process r_{i+1}. Consider the case of conflicts. The automaton looks for a marked event (e_i, m_i) with $\mathsf{rank}(\mathsf{e}_i) = \mathsf{r}_i$ marked by $\mathsf{leave} \in m_i$. It remembers the kind and the address of this event. Then, it seeks a marked event $(\mathsf{e}_{i+1}, m_{i+1})$ with $\mathsf{rank}(\mathsf{e}_{i+1}) = \mathsf{r}_{i+1}$ marked by $\mathsf{enter} \in m_{i+1}$. If both events are found, they touch the same address, and one of them is a write, the automaton reaches the accepting state. Since finite automata are closed under intersection, we can define the *finite automaton of cycle type* cyc as $Z^{\mathsf{cyc}} := Z^{\mathsf{r}_1,\mathsf{r}_2} \cap \ldots \cap Z^{\mathsf{r}_{k-1},\mathsf{r}_k} \cap Z^{\mathsf{r}_k,\mathsf{r}_1}$.

▶ **Theorem 16.** *$\mathcal{P}$ is robust iff $\mathcal{L}(Y^{\mathsf{M}}(\mathcal{P}, N)) \cap \mathcal{L}(Z^{\mathsf{cyc}}) = \emptyset$ for all cycle types cyc.*

We can now prove Theorem 3. To check whether $(\mathcal{P}, N)$ is robust, we go over all cycle types $\mathsf{cyc} = \mathsf{r}_1 \dots \mathsf{r}_k$. This enumeration of cycle types can be done in space that is polynomial in N. For each such sequence, we check if $\mathcal{L}(Y^{\mathsf{M}}(\mathcal{P}, N)) \cap \mathcal{L}(Z^{\mathsf{cyc}}) = \emptyset$. By Theorem 16, the program is robust iff all intersections are empty. By Lemma 10, there is a 4-headed finite state automaton W with $\mathcal{L}(W) = \mathcal{L}(Y^{\mathsf{M}}(\mathcal{P}, N)) \cap \mathcal{L}(Z^{\mathsf{cyc}})$. Since the size of W is exponential in the size of $(\mathcal{P}, N)$ and emptiness is in NL by Lemma 11, deciding $\mathcal{L}(W) = \emptyset$ can be done in space that is polynomial in $(\mathcal{P}, N)$. This shows robustness is in PSpace.

References

1 S. V. Adve and M. D. Hill. A unified formalization of four shared-memory models. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):613–624, 1993.

2 J. Alglave. *A Shared Memory Poetics*. PhD thesis, University Paris 7, 2010.

3 J. Alglave and L. Maranget. Stability in weak memory models. In *CAV*, volume 6806 of *LNCS*, pages 50–66. Springer, 2011.

4 D. Bonachea. GASNet specification, v1.1. Technical Report UCB/CSD-02-1207, University of California, Berkeley, 2002.

5 A. Bouajjani, E. Derevenetc, and R. Meyer. Checking and enforcing robustness against TSO. In *ESOP*, volume 7792 of *LNCS*, pages 533–553. Springer, 2013.

6 A. Bouajjani, R. Meyer, and E. Möhlmann. Deciding robustness against Total Store Ordering. In *ICALP*, volume 6756 of *LNCS*, pages 428–440. Springer, 2011.

7 S. Burckhardt and M. Musuvathi. Effective program verification for relaxed memory models. In *CAV*, volume 5123 of *LNCS*, pages 107–120. Springer, 2008.

8 J. Burnim, C. Stergiou, and K. Sen. Sound and complete monitoring of sequential consistency for relaxed memory models. In *TACAS*, volume 6605 of *LNCS*, pages 11–25. Springer, 2011.

9 G. Calin, E. Derevenetc, R. Majumdar, and R. Meyer. A theory of partitioned global address spaces. *CoRR*, abs/1307.6590, 2013. `http://arxiv.org/abs/1307.6590`.

10 B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith. Introducing OpenSHMEM: SHMEM for the PGAS community. In *PGAS*, page 2. ACM, 2010.

11 UPC Consortium. UPC language specification v1.2. Technical report, 2005.

12 J. Dassow and G. Păun. *Regulated Rewriting in Formal Language Theory*, volume 18 of *Monographs in Theoretical Computer Science. An EATCS Series*. Springer, 1989.

13 D. Dice. A race in locksupport park() arising from weak memory models. `https://blogs.oracle.com/dave/entry/a_race_in_locksupport_park`, Nov 2009.

14 J. Dinan, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur. An implementation and evaluation of the MPI 3.0 one-sided communication interface. `www.mcs.anl.gov/uploads/cels/papers/P4014-0113.pdf`.

15 Global address space programming interface. `http://www.gaspi.de/`.

16 P. N. Hilfinger, D. O. Bonachea, K. Datta, D. Gay, S. L. Graham, B. R. Liblit, G. Pike, J. Zh. Su, and K. A. Yelick. Titanium language reference manual, version 2.19. Technical Report UCB/EECS-2005-15, UC Berkeley, 2005.

17 D. Kozen. Lower bounds for natural proof systems. In *FOCS*, pages 254–266. IEEE, 1977.

18 L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979.

19 R. Machado and C. Lojewski. The Fraunhofer virtual machine: a communication library and runtime system based on the RDMA model. *Computer Science — Research and Development*, 23(3-4):125–132, 2009.

20 A. Muzahid, S. Qi, and J. Torrellas. Vulcan: Hardware support for detecting sequential consistency violations dynamically. In *MICRO*, pages 363–375. IEEE, 2012.

21 J. Nieplocha and B. Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. In *Parallel and Distributed Processing*, volume 1586 of *LNCS*, pages 533–546. Springer, 1999.

22 The UPC NAS parallel benchmarks. `http://upc.gwu.edu/download.html`.

23 R. W. Numrich and J. Reid. Co-array Fortran for parallel programming. In *ACM Sigplan Fortran Forum*, volume 17, pages 1–31. ACM, 1998.

24 S. Owens. Reasoning about the implementation of concurrency abstractions on x86-TSO. In *ECOOP*, volume 6183 of *LNCS*, pages 478–503. Springer, 2010.

25 C.-S. Park, K. Sen, P. Hargrove, and C. Iancu. Efficient data race detection for distributed memory parallel programs. In *SC'11*, page 51. ACM, 2011.

26 D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM TOPLAS*, 10(2):282–312, 1988.