

# **Modeling Synchronization and Consistency for Data Race Detection in Remote Memory Access Programs**

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der  
RWTH Aachen University zur Erlangung des akademischen Grades  
eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

**Simon Schwitanski, Master of Science**

aus Kempen

Berichter: Univ.-Prof. Dr. rer. nat. Matthias S. Müller  
Univ.-Prof. Dr. rer. nat. Martin Schulz

Tag der mündlichen Prüfung: 05. Februar 2025

Diese Dissertation ist auf den Internetseiten der Universitätsbibliothek online verfügbar.



# Abstract

The increasing parallelism in today's supercomputers requires scalable parallel programming methods with efficient communication models. The traditional communication model in scientific computing is message passing: Both the sending and the receiving processes participate actively in the data exchange via messages. Remote Memory Access (RMA) models provide an alternative communication method where processes can access the memory of other processes directly. RMA models avoid unnecessary synchronization between processes and outperform the traditional message-passing model in modern supercomputers. However, they require users to explicitly ensure the synchronization and consistency of memory accesses through corresponding API calls. Otherwise, concurrent conflicting memory accesses lead to data races with undefined behavior. The non-deterministic nature of data races, commonly known from shared-memory programming, makes their manual detection difficult.

This thesis investigates data races in RMA programs and provides novel scalable methods to detect them at runtime. A classification of data races in the RMA models MPI RMA, OpenSHMEM, and GASPI shows that synchronization and consistency are the two key properties that a correctness tool must capture to identify RMA races. This thesis provides formal models that allow analyzing both properties in RMA programs. The synchronization model analyzes the happened-before relation of events using a vector clock exchange. It captures the synchronization state of processes at runtime. The consistency model formalizes a relation defining when a remote memory access is guaranteed to be completed. Both models are combined in a generalized on-the-fly race detection model that can detect RMA data races independent of the concrete RMA model used in an application.

The developed race detection model is implemented in a tool named RMA Sanitizer. It combines the shared-memory race detector ThreadSanitizer with the correctness checking tool MUST to detect RMA data races in MPI RMA, OpenSHMEM, and GASPI at runtime. For the evaluation, this thesis provides RMA Race Bench, a classification quality benchmark suite designed to quantify the detection accuracy of RMA race detection tools. The evaluation with RMA Race Bench shows that RMA Sanitizer has the highest detection accuracy compared to other state-of-the-art RMA race detectors. An overhead study with RMA proxy applications running with more than 700 processes shows that RMA Sanitizer is applicable to large-scale workloads.



# Kurzfassung

Der stetig wachsende Parallelismus in heutigen Supercomputern erfordert skalierbare, parallele Programmiermethoden mit effizienten Kommunikationsmodellen. Das traditionelle Kommunikationsmodell im wissenschaftlichen Rechnen ist der Nachrichtenaustausch: Sowohl der sendende als auch der empfangende Prozess ist aktiv am Datenaustausch über Nachrichten beteiligt. Remote-Memory-Access-Modelle (RMA) stellen eine alternative Kommunikationsmethode bereit, in der Prozesse direkt auf den Speicher von anderen Prozessen zugreifen. RMA-Modelle vermeiden unnötige Synchronisation zwischen Prozessen und erzielen in modernen Supercomputern eine bessere Leistung als der klassische Nachrichtenaustausch. Sie erfordern jedoch, dass der Nutzer explizit Synchronisation und Konsistenz der Speicherzugriffe durch entsprechende API-Aufrufe sicherstellt. Andernfalls führen gleichzeitige, im Konflikt stehende Speicherzugriffe zu Data Races mit undefiniertem Verhalten. Der Nichtdeterminismus von Data Races, bekannt von Shared-Memory-Programmierung, macht ihre manuelle Erkennung komplex.

Diese Arbeit untersucht Data Races in RMA-Programmen und präsentiert neue skalierbare Methoden, um diese zur Laufzeit zu erkennen. Eine Klassifikation von Data Races in den RMA-Modellen MPI RMA, OpenSHMEM und GASPI zeigt, dass Synchronisation und Konsistenz die wesentlichen Eigenschaften zur Data-Race-Erkennung in RMA sind, die ein Korrektheitsanalysewerkzeug untersuchen muss. Diese Arbeit definiert formale Modelle, die die Analyse von beiden Eigenschaften in RMA-Programmen ermöglichen. Das Synchronisationsmodell analysiert die Happened-Before-Relation von Ereignissen mithilfe eines Vektoruhraustauschs. Dieser zeichnet den Synchronisationszustand von Prozessen zur Laufzeit auf. Das Konsistenzmodell formalisiert eine Relation, die definiert, wann ein RMA-Zugriff garantiert abgeschlossen ist. Beide Modelle werden in einem generalisierten Modell zur Laufzeiterkennung von Data Races in RMA-Programmen, unabhängig vom konkret genutzten RMA-Modell, kombiniert.

Das entwickelte Modell zur Erkennung von Data Races ist im Werkzeug RMASanitizer implementiert. Es kombiniert ThreadSanitizer, ein Werkzeug zur Erkennung von Data Races in Shared-Memory-Programmen, mit dem Korrektheitsanalysewerkzeug MUST, um Data Races in MPI RMA, OpenSHMEM und GASPI zur Laufzeit zu erkennen. Zur Evaluation wird RMARaceBench entwickelt, ein Benchmark zur Analyse der Klassifikationsqualität, der die Erkennungsgenauigkeit von Werkzeugen zur Race-Erkennung in RMA quantifiziert. Die Evaluation mit RMARaceBench zeigt, dass RMASanitizer die höchste Erkennungsgenauigkeit verglichen mit anderen Werkzeugen hat. Eine Overheaduntersuchung mit RMA-Proxy-Applikationen, die mit mehr als 700 Prozessen ausgeführt werden, zeigt, dass RMASanitizer in hochskalierenden Anwendungen nutzbar ist.



# Contents

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Introduction</b>   | <b>1</b>  |
| 1.1      | Motivation . . . . .  | 2         |
| 1.2      | Contributions . . . . .   | 3         |
| 1.3      | Thesis Structure . . . . .  | 4         |
| <b>2</b> | <b>Data Races in RMA Programs</b>   | <b>7</b>  |
| 2.1      | PGAS and RMA Programming Models . . . . .                                   | 7         |
| 2.1.1    | MPI RMA . . . . .   | 9         |
| 2.1.2    | OpenSHMEM . . . . .   | 9         |
| 2.1.3    | GASPI . . . . .   | 10        |
| 2.2      | RMA Programming Fundamentals . . . . .                                      | 10        |
| 2.2.1    | Memory Management and Models . . . . .                                      | 11        |
| 2.2.2    | Communication Routines . . . . .  | 13        |
| 2.2.3    | Completion and Synchronization Routines . . . . .                           | 13        |
| 2.3      | RMA Race Classification . . . . .   | 18        |
| 2.3.1    | Data Race Terminology . . . . .   | 19        |
| 2.3.2    | Local Buffer Races . . . . .  | 19        |
| 2.3.3    | Remote Races . . . . .  | 21        |
| 2.3.4    | Incorrect Atomicity . . . . .   | 23        |
| 2.3.5    | Hybrid Races . . . . .  | 24        |
| 2.3.6    | Benign Races . . . . .  | 25        |
| 2.3.7    | Related Work . . . . .  | 26        |
| 2.4      | Results and Discussion . . . . .  | 27        |
| <b>3</b> | <b>Clock-Based Synchronization Analysis for Distributed-Memory Programs</b> | <b>29</b> |
| 3.1      | Happened-Before Relation and Vector Clocks . . . . .                        | 30        |
| 3.2      | Synchronization in Distributed-Memory Programs . . . . .                    | 32        |
| 3.2.1    | Process-Bound Synchronization . . . . .                                     | 34        |
| 3.2.2    | Resource-Bound Synchronization . . . . .                                    | 35        |
| 3.3      | Generalized Vector Clock Exchange . . . . .                                 | 37        |
| 3.3.1    | Point-to-Point Synchronization . . . . .                                    | 39        |
| 3.3.2    | Collective Synchronization . . . . .  | 41        |
| 3.3.3    | Locks and Polling . . . . .   | 45        |
| 3.3.4    | Non-Deterministic Synchronization . . . . .                                 | 46        |
| 3.3.5    | Limitations . . . . .   | 49        |
| 3.4      | Scalable On-the-Fly Tracking . . . . .                                      | 50        |
| 3.4.1    | Call Interception . . . . .   | 50        |
| 3.4.2    | Communication Layer . . . . .   | 51        |

## Contents

|          |  |            |
|----------|--|------------|
| 3.4.3    | Piggybacking Vector Clocks . . . . .                     | 53         |
| 3.4.4    | Tracking Handles With MUST . . . . .                     | 53         |
| 3.4.5    | Analysis Workflow . . . . .                              | 54         |
| 3.4.6    | User Annotations . . . . .                               | 56         |
| 3.4.7    | Summary . . . . .  | 57         |
| 3.5      | Overhead Evaluation . . . . .                            | 57         |
| 3.5.1    | Experiment Setup . . . . .                               | 58         |
| 3.5.2    | Results and Discussion . . . . .                         | 58         |
| 3.6      | Hybrid Parallelism . . . . .                             | 60         |
| 3.7      | Use Cases . . . . .                                      | 62         |
| 3.7.1    | MPI I/O Race Detection . . . . .                         | 62         |
| 3.7.2    | Thread-Level Concurrency Checks for MPI+OpenMP . . . . . | 64         |
| 3.8      | Related Work . . . . .                                   | 66         |
| 3.8.1    | Shared-Memory Race Detection . . . . .                   | 66         |
| 3.8.2    | Alternative MPI Matchings . . . . .                      | 67         |
| 3.8.3    | Post-Mortem RMA Race Detection . . . . .                 | 67         |
| 3.8.4    | Other Related Work . . . . .                             | 68         |
| 3.9      | Results and Discussion . . . . .                         | 69         |
| <b>4</b> | <b>Event-Based RMA Race Detection Model</b>              | <b>71</b>  |
| 4.1      | Consistency Model for RMA Programs . . . . .             | 72         |
| 4.1.1    | Event Model . . . . .                                    | 73         |
| 4.1.2    | Consistency Inference Rules . . . . .                    | 75         |
| 4.1.3    | Data Race Formalization . . . . .                        | 80         |
| 4.2      | RMA Race Detection with Concurrent Regions . . . . .     | 80         |
| 4.2.1    | Local Buffer Races . . . . .                             | 81         |
| 4.2.2    | Remote Races . . . . .                                   | 82         |
| 4.2.3    | Race Detection Algorithm . . . . .                       | 89         |
| 4.3      | Limitations . . . . .                                    | 91         |
| 4.3.1    | Model-Specific Concepts . . . . .                        | 91         |
| 4.3.2    | Separate Memory Model in MPI RMA . . . . .               | 92         |
| 4.3.3    | Hybrid Parallelism . . . . .                             | 92         |
| 4.4      | Generalizability . . . . .                               | 93         |
| 4.5      | Related Work . . . . .                                   | 94         |
| 4.5.1    | MPI RMA Axiomatic Model . . . . .                        | 94         |
| 4.5.2    | coreRMA Language . . . . .                               | 95         |
| 4.5.3    | Graph-Based Modeling of RMA Programs . . . . .           | 96         |
| 4.5.4    | State-Space Automata for PGAS Semantics . . . . .        | 97         |
| 4.5.5    | SPMD IR . . . . .  | 97         |
| 4.5.6    | Concurrency Intermediate Verification Language . . . . . | 98         |
| 4.5.7    | Race Detection Models . . . . .                          | 99         |
| 4.6      | Results and Discussion . . . . .                         | 100        |
| <b>5</b> | <b>Scalable Race Detection for RMA Programs</b>          | <b>101</b> |
| 5.1      | Related Work on RMA Race Detection . . . . .             | 102        |
| 5.1.1    | Post-Mortem DAG Traversal . . . . .                      | 102        |
| 5.1.2    | Post-Mortem Timestamping . . . . .                       | 103        |

|          |   |            |
|----------|---|------------|
| 5.1.3    | Post-Mortem Task Graphs . . . . .   | 103        |
| 5.1.4    | On-the-Fly Detection for Bulk-Synchronous Programs . . . . .              | 103        |
| 5.1.5    | On-the-Fly Mirror Windows . . . . .                                       | 105        |
| 5.1.6    | Active Testing . . . . .  | 105        |
| 5.1.7    | Static Analysis . . . . .   | 106        |
| 5.1.8    | Model Checking Approaches . . . . .                                       | 107        |
| 5.1.9    | Nasty-MPI . . . . .   | 107        |
| 5.1.10   | Discussion . . . . .  | 107        |
| 5.2      | Shared-Memory Race Detection with ThreadSanitizer . . . . .               | 108        |
| 5.2.1    | Instrumentation . . . . .   | 108        |
| 5.2.2    | State Machine . . . . .   | 109        |
| 5.2.3    | User Annotations . . . . .  | 111        |
| 5.2.4    | Fibers . . . . .  | 111        |
| 5.3      | RMA Sanitizer Architecture . . . . .                                      | 113        |
| 5.3.1    | Selective Local Memory Access Instrumentation . . . . .                   | 114        |
| 5.3.2    | Synchronization and Consistency Tracking . . . . .                        | 118        |
| 5.3.3    | Concurrent Region Detection . . . . .                                     | 120        |
| 5.3.4    | Concurrent Regions as ThreadSanitizer Fibers . . . . .                    | 121        |
| 5.3.5    | Race Detection Workflow . . . . .   | 124        |
| 5.3.6    | Alternative Access Tracking with Interval Skip Lists . . . . .            | 128        |
| 5.3.7    | Limitations . . . . .   | 130        |
| 5.4      | Results and Discussion . . . . .  | 131        |
| <b>6</b> | <b>Classification Quality and Overhead Analysis of RMA Race Detection</b> | <b>133</b> |
| 6.1      | Classification Quality with RMA RaceBench . . . . .                       | 134        |
| 6.1.1    | Methodology . . . . .   | 134        |
| 6.1.2    | Related Benchmark Suites . . . . .  | 138        |
| 6.1.3    | RMA RaceBench Results . . . . .   | 140        |
| 6.1.4    | Discussion . . . . .  | 148        |
| 6.2      | Overhead Analysis . . . . .   | 149        |
| 6.2.1    | Applications . . . . .  | 150        |
| 6.2.2    | Setup . . . . .   | 151        |
| 6.2.3    | Overhead Results . . . . .  | 153        |
| 6.2.4    | Comparison to PARCOACH . . . . .  | 158        |
| 6.3      | Results and Discussion . . . . .  | 161        |
| <b>7</b> | <b>Summary and Conclusion</b>   | <b>163</b> |
| 7.1      | Future Work . . . . .   | 165        |
| <b>8</b> | <b>Statement of Originality</b>   | <b>167</b> |
| <b>A</b> | <b>Thesis Artifact</b>  | <b>171</b> |
| <b>B</b> | <b>RMA RaceBench Results</b>  | <b>173</b> |
| B.1      | MPI RMA Results . . . . .   | 173        |
| B.2      | SHMEM Results . . . . .   | 175        |
| B.3      | GASPI Results . . . . .   | 177        |

*Contents*

|                        |            |
|------------------------|------------|
| <b>Bibliography</b>    | <b>181</b> |
| <b>List of Figures</b> | <b>193</b> |
| <b>List of Tables</b>  | <b>195</b> |

# 1 Introduction

Over the last decades, computer simulations have become the third pillar of science, besides theory and experiments. They can answer questions that would be too time-intensive, costly, or impossible to investigate with real experiments. Supercomputers consisting of many interconnected processors made many large-scale simulations possible in the first place. Application areas, among many others, include weather forecasts and climate simulation in meteorology, flow and material simulation in engineering, genome sequencing in bioinformatics, or training and inference of machine learning models.

For several decades until around 2005, chip manufacturers achieved performance improvements of processors through clock rate growths. However, the increased power consumption and resulting high heat dissipation, along with higher clock rates, made further improvements in this area difficult [30]. Therefore, multi-core architectures consisting of multiple processing entities on a single integrated circuit became predominant. This shift in the hardware architecture led to new requirements in the software architecture: The ever-growing number of parallel execution units and the rapidly evolving hardware has to be utilized efficiently to profit from them. Parallel programming models are a fundamental building block towards that goal. The Message Passing Interface (MPI) [28], which was standardized in 1994, is the de-facto standard approach of parallel programming in supercomputers across compute nodes, also referred to as *distributed-memory parallelism*. Processes can independently compute intermediate results of a larger computation and exchange information with other processes by passing messages to each other. For parallelism within a compute node, OpenMP [7] has emerged as a parallel programming model, also referred to as *shared-memory parallelism*: Threads can run in parallel and directly access shared memory.

In MPI, the classical communication model via messages is *two-sided*: Both the sending and receiving process are actively involved in the data exchange. This kind of communication proved to be quite successful for many scientific parallel algorithms. However, for applications with irregular communication patterns, such as graph algorithms, the sender-receiver approach of MPI leads to suboptimal performance [46, 60]. Moreover, message-passing communication does not match the hardware architecture of Remote Direct Memory Access (RDMA) networks used in today's supercomputers [38]. To overcome those limitations on the software side, *Remote Memory Access* (RMA) and the related *Partitioned Global Address Space* (PGAS) programming models have been developed: Using those models, processes in a distributed-memory system can directly access the memory of remote processes through reads and writes. Such communication is

also often named *one-sided* because the targeted remote process is not actively involved in data exchange compared to two-sided communication via message passing. Examples of such RMA programming models include MPI RMA [28, §12], OpenSHMEM [19], and GASPI [27]. RMA models have been utilized successfully in various graph algorithms [46, 60], CFD applications [98] and quantum chemistry applications [96]. Recently, RMA models such as NVSHMEM have been developed specifically for efficient intra-kernel communication between GPUs [40].

### 1.1 Motivation

While RMA models offer performance advantages over traditional two-sided MPI communication, they also come with a new challenge: Their complex API and semantics make writing correct RMA programs difficult. Since processes can remotely access the memory of other processes, appropriate coordination of accesses is required to ensure correctness. For that, users have fine-granular control over the synchronization and consistency behavior of remote memory accesses. *Synchronization* describes when a process actively waits for a signal of another process, establishing a partially ordered execution of events. *Consistency* describes when a memory access is guaranteed to be finished. Concurrent execution of conflicting memory accesses without properly established synchronization and consistency leads to *data races* in RMA programs, a commonly known problem from shared-memory programs, resulting in undefined behavior. The non-deterministic nature of data races makes their manual detection difficult. Therefore, appropriate support through tools detecting data races and pointing to the affected source code locations is desirable.

Data races in shared-memory programs are a well-studied problem [9, 67], so mature correctness tools are available [3, 26, 77, 94, 95] which analyze and report races during the runtime of the application. However, translating the detection methods of shared-memory race detectors to RMA programs is challenging for several reasons: First, the consistency semantics of RMA models are more complex to analyze than those of shared-memory programs. Second, the various ways of synchronization in distributed-memory systems have to be correctly understood by a tool. Third, for runtime analysis tools, the execution environment of a distributed-memory system makes the efficient exchange of required analysis information difficult. Thus, RMA race detection tools and algorithms must cope with those additional challenges.

Existing work on RMA race detection provides static [82], on-the-fly [2], and post-mortem [16, 22, 49, 73] analysis algorithms and tools. However, those approaches only support a small subset of the synchronization and consistency mechanisms provided in RMA and are often not scalable in practice. Further, existing work typically focuses on a single RMA model. Since there is no de-facto standard model for RMA programming and there is a high amount and variety of RMA models [21] with different syntax and semantics used in practice, the general applicability of existing tools is limited.

This thesis addresses the understanding, modeling, and detection of data races in RMA applications while fully covering all possible completion and synchronization modes in RMA. It classifies data races in RMA models and defines a generic race detection model suited for on-the-fly RMA data race detection. The race detection model consists of two essential components: First, a synchronization model based on vector clocks [87] to understand the concurrency of events RMA programs. Second, a consistency model to understand the completion behavior of remote memory accesses and the interaction with local memory accesses. The model’s generalizability is showcased by applying it to three state-of-the-art RMA models: MPI RMA, OpenSHMEM, and GASPI. The model’s practicability is showcased by its implementation in an on-the-fly RMA race detector named RMASanitizer.

## 1.2 Contributions

The key contributions of my thesis focus on modeling and analyzing the semantics of Remote Memory Access (RMA) programs for data race detection, focusing on MPI RMA, OpenSHMEM, and GASPI. The contributions include defining formal models to capture the synchronization and consistency of accesses in RMA programs and implementing those models in a scalable RMA race detector, named RMASanitizer [93]. The four major contributions are further outlined in the following.

First, this thesis classifies the different data races that may occur in RMA programs. Compared to shared-memory programming, the various ways of synchronization in RMA and the relaxed memory consistency make writing race-free RMA programs more difficult. This thesis systematically analyzes and compares the standards of state-of-the-art RMA programming models to understand and classify all possible data race scenarios. To that end, I have developed a benchmark suite, named *RMARaceBench* [91], consisting of a large set of test programs that all exhibit different data races, covering the RMA models MPI RMA, OpenSHMEM, and GASPI. The evaluation part of this thesis uses *RMARaceBench* to evaluate and compare different RMA race detectors systematically.

Second, this thesis provides a clock-based model to analyze the synchronization in distributed-memory programs, as reasoning on data races requires understanding the execution order of events. The model uses the well-known vector clocks [24, 87] to capture the synchronization states between the processes. It is designed to capture the different synchronization methods in distributed-memory programming, mainly covering the RMA models MPI RMA, OpenSHMEM, and GASPI. The model defines a vector clock exchange for generic synchronization primitives and maps the concrete routines in RMA programming models to the corresponding primitives. As stated before, previous approaches only covered certain kinds of synchronization for a specific programming model. Based on my proposed model, I have implemented a scalable vector clock exchange [89] that tracks the synchronization state of processes in RMA programs at runtime. The implementation is subsequently used for the RMA race detection in RMASanitizer.

Third, a formal model [90, 93] suitable for on-the-fly data race detection in RMA programs is presented. The model abstracts the specific RMA routines affecting the memory consistency in MPI RMA, OpenSHMEM, and GASPI to generic primitives. Although most consistency semantics are similar in all RMA models, there are certain specific corner cases in each model that also have to be considered in a unified abstract model. The consistency model is combined with the clock-based synchronization model to define an on-the-fly race detection algorithm. The proposed algorithm is the first approach that uses vector clocks for on-the-fly race detection in RMA programs.

Fourth, the applicability of the concepts is implemented in a production-ready RMA race detector named RMASanitizer [93]. The tool performs an on-the-fly detection of data races in MPI RMA, OpenSHMEM, and GASPI programs. It combines static and dynamic program analysis principles to achieve high detection accuracy while delivering reasonable overhead, even for large-scale program runs. As the evaluation in this thesis shows, RMASanitizer’s detection accuracy is the highest among other RMA race detectors. An overhead evaluation with RMA proxy applications running with over 700 processes shows that it can be applied to large-scale programs. RMASanitizer relies on the infrastructure of the correctness checking tool MUST [36]. It is open-sourced as part of the thesis artifact described in Appendix A.

### 1.3 Thesis Structure

In the following, Chapter 2 provides background knowledge on RMA models, particularly MPI RMA, OpenSHMEM, and GASPI. This includes the fundamental concepts of RMA programming, such as memory management, communication, completion, and synchronization routines. Further, it defines the term “data race” in the context of RMA programming, classifies the different race situations possible in RMA models, and illustrates them with code examples.

Chapter 3 introduces the clock-based model to capture the synchronization in distributed-memory applications. First, the synchronization concepts present in MPI, OpenSHMEM, and GASPI are abstracted into a set of synchronization primitives. Then, a generalized vector clock exchange based on those primitives is discussed. The chapter also presents an implementation of a scalable on-the-fly tracking of vector clocks based on the correctness checking tool MUST [36]. Its overhead is evaluated based on the SPEC MPI 2007 benchmarks. Finally, other use cases besides RMA race detection, such as MPI I/O race detection and thread-level concurrency checks for MPI+OpenMP programs, are discussed.

In Chapter 4, the generic race detection model for RMA applications is presented. It formalizes a consistency model for RMA programs and combines it with the clock-based synchronization model discussed in Chapter 3 to derive concurrent regions of RMA operations, representing the period in which the underlying memory access may take

place. The race detection model is applied to different race examples to showcase its application. The chapter concludes with a discussion of the model's generalizability and related approaches.

Chapter 5 presents an implementation of the race detection model in a scalable on-the-fly race detector named *RMASanitizer*. It combines the shared-memory ThreadSanitizer [95] with the correctness checking tool MUST [36] to perform RMA race detection in MPI RMA, OpenSHMEM, and GASPI programs. The chapter first discusses related approaches to RMA race detection and introduces ThreadSanitizer's detection approach for shared-memory race detection. Then, the chapter presents the architecture of *RMASanitizer* in detail, including its instrumentation techniques, the detection of concurrent regions of RMA operations, and the interfacing with ThreadSanitizer.

In Chapter 6, the classification quality and the overhead of *RMASanitizer* is evaluated. The chapter first presents the methodology of the benchmark suite *RMARaceBench* that can evaluate the data race detection accuracy of RMA race detectors. Then, the results of *RMASanitizer* and other RMA race detection approaches are evaluated with *RMARaceBench* and compared. Moreover, the results of the overhead evaluation of *RMASanitizer* on different RMA proxy applications are presented to showcase its applicability to real-world codes.

Chapter 7 summarizes the results of this thesis and provides an outlook of future extensions to the proposed synchronization, consistency, and race detection models, and the race detector *RMASanitizer*.



## 2 Data Races in RMA Programs

Data races are a common problem that arises in parallel programs and leads to undefined behavior. A data race [67] occurs in a parallel program execution between two memory accesses if (1) both address the same memory location, (2) at least one of them is a write access, (3) they are executed by different parallel entities, and (4) they are improperly synchronized. Typically known from shared-memory programming, data races also occur in RMA since processes can access other processes' memory directly. First, this chapter gives an overview of RMA programming models in Section 2.1 and the basic primitives used in RMA programs in Section 2.2, focusing on MPI RMA [28], OpenSHMEM [19], and GASPI [27]. Second, a classification of races in RMA models is presented in Section 2.3, distinguishing local buffer races and remote races as primary classes.

### 2.1 PGAS and RMA Programming Models

In parallel programming, the traditional way of communication in distributed-memory computers is message-passing using the Message Passing Interface (MPI) [28] where processes explicitly exchange data by sending and receiving messages. The memory remains private to each process. In a shared-memory computer, threads within a single process perform read and write accesses to a shared memory region. They have to synchronize via critical sections, barriers, or locks, to ensure that exchanged data has arrived correctly and avoid concurrent conflicting accesses. Such parallelism is utilized by the shared-memory parallel programming model OpenMP [7].

While shared-memory programming via OpenMP provides implicit communication via reads and writes to memory, it cannot scale beyond a single compute node to larger clusters. On the other hand, distributed-memory communication via message passing scales to large clusters with many compute nodes, but requires both the sending and the receiving process to be actively involved – via MPI function calls – in the message exchange. This may penalize performance, particularly for applications with irregular communication patterns [46]. Partitioned Global Address Space (PGAS) and Remote Memory Access (RMA) programming models combine the concepts of shared-memory and distributed-memory models to get the best out of both worlds [21]: In PGAS and RMA, any process in a distributed-memory environment can access the memory of any target process *directly*. For that, the basic remote memory access primitives *put* and *get* are defined, representing a remote write to another process or a remote read from

## 2 Data Races in RMA Programs

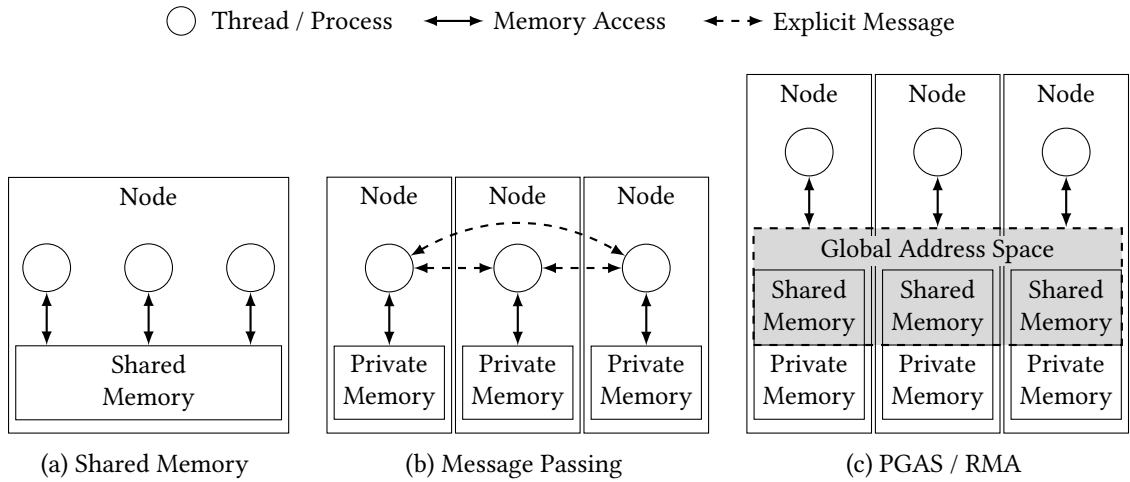


Figure 2.1: High-level view of the memory models used in shared memory, message passing, and PGAS/RMA. PGAS provides a global address space that can be accessed from all processes across compute nodes. Adapted from [32].

another process, respectively. The communication via such remote memory accesses is sometimes also referred to as *one-sided communication*, since the sending process specifies all communication parameters without the target process being involved [28]. Compared to two-sided message passing, this decouples communication and the implicit synchronization of the sending and receiving process. Further, the remote memory access primitives in PGAS and RMA directly map to the Remote Direct Memory Access (RDMA) functionality of modern interconnects [38]. Figure 2.1 illustrates the conceptual differences between shared memory, message passing, and PGAS / RMA programming.

In PGAS programming [21], the global address space is partitioned between the processes, where each process contributes a local fraction of its private memory to the address space. A process can access its local memory faster than memory locations residing at other processes due to the additional network latency. In PGAS terminology, the memory regions residing at different processes are often called *places*. There are different approaches to model the costs of local and remote memory accesses to and from those places [21].

The terms PGAS and RMA are often used interchangeably in the literature, since both refer to a model enabling access to the memory of other processes. However, they address slightly different concepts, as discussed by Hoefler et al. [38]: The term “PGAS” focuses on a shared-memory abstraction in a global address space. It does not explicitly specify the underlying communication when an element in the global address space is accessed. On the other hand, the term “RMA” focuses on the access primitives, e.g., put and get, closer to the system level to exchange data between processes. RMA comprises any programming model that provides remote access primitives and, therefore, a wider class of programming models than PGAS. Due to the focus on remote memory access primitives and their semantics, this thesis uses the term RMA more commonly than PGAS.

PGAS and RMA have been implemented in various ways in the past: Language-based extensions such as Coarray Fortran [69], Unified Parallel C [31], and Titanium [110] add new language concepts that either implicitly or explicitly allow for remote memory accesses to a global address space. Other approaches define a new programming language, such as Chapel [14] or X10 [15], which – beside PGAS – introduce an asynchronous execution model with tasking. Further, the directive-based PGAS approach XcalableMP [58] relies on compiler pragmas added to C or Fortran code to parallelize certain parts of the execution, similar to OpenMP but focused on distributed-memory systems. Lastly, there are library-based approaches such as MPI RMA [28, §12], OpenSHMEM [19], GASPI [27], UPC++ [4], ARMCI [68], GASNet [10], and DASH [29] that provide designated functions to allocate remotely accessible memory regions and perform remote memory accesses. Most of the library-based approaches are more explicit and operate closer to the system level than the language-based approaches: Instead of a highly abstracted global array view that is resolved to remote memory accesses implicitly, the user has to explicitly specify the remote memory access via *put* and *get* primitives. Higher-level PGAS languages often rely on lower-level RMA libraries. For example, Coarray Fortran, UPC, and Chapel internally use GASNet in their implementation.

This thesis focuses on the library-based RMA models provided by MPI RMA, OpenSHMEM, and GASPI, which are shortly outlined in the following paragraphs.

### 2.1.1 MPI RMA

MPI Remote Memory Access (MPI RMA) is part of the MPI library specification [28, §12] since MPI-2 in 1997 and defines basic primitives *MPI\_Put* and *MPI\_Get* to directly access the memory of other MPI processes, without requiring the target process being actively involved, as shown in Figure 2.2. MPI RMA relies on a Single Program Multiple Data (SPMD) programming style and supports C and Fortran. All MPI RMA primitives are compatible with any other MPI call, i.e., point-to-point messaging and collectives can be used together with RMA in the same MPI program. Memory regions have to be explicitly exposed by a process to be accessible with RMA primitives. In terms of synchronization, MPI RMA supports various modes ranging from bulk-synchronous to fine-grained individual completion, which will be introduced in Section 2.2.3.

### 2.1.2 OpenSHMEM

OpenSHMEM [19], also known as SHMEM, is another library specification for RMA, originally started as Cray SHMEM and SGI SHMEM in 1993 and was subsequently standardized as OpenSHMEM in 2012. Similar to MPI RMA, it uses an SPMD execution model and defines basic access primitives to read and write the memory of other SHMEM processes. It only provides a C/C++ interface, the Fortran interface was deprecated in OpenSHMEM 1.4 (2017) and finally removed in OpenSHMEM 1.5 (2020). Different

## 2 Data Races in RMA Programs

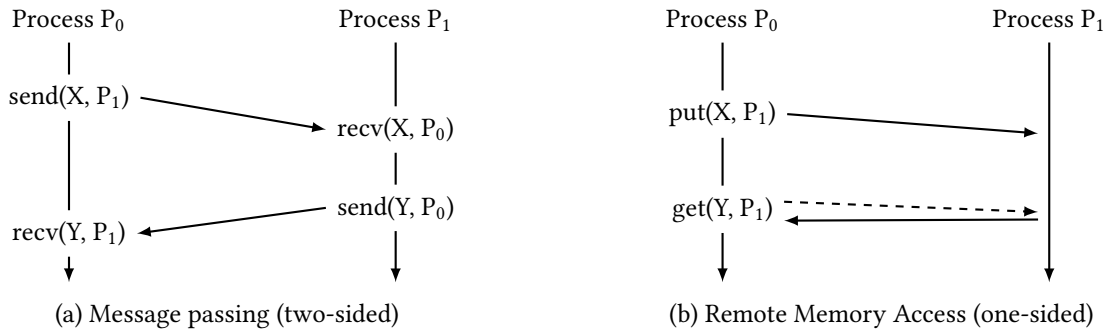


Figure 2.2: Message passing (two-sided) and Remote Memory Access (one-sided) communication. In one-sided communication, process P<sub>0</sub> calls *put* and *get* to remotely write to and read from P<sub>1</sub>, while P<sub>1</sub> itself stays completely passive.

from MPI RMA, the memory model consists of *symmetric data objects*, i.e., all remotely accessible memory regions have the same size on each process.

### 2.1.3 GASPI

The Global Address Space Programming Interface (GASPI) [27] is also an RMA library specification for C/C++ and Fortran programs that originally started in 2011 as a joint effort of German research institutions to define an API for PGAS programs. Its main difference to MPI RMA and SHMEM is that the memory model supports heterogeneous memory architectures, i.e., allocated memory segments for remote memory accesses may target the CPU's main memory or memory segments of a GPGPU, coprocessor, or any other non-volatile memory. Another difference is that the specification is designed around fault tolerance such that failing processes can be detected by timeouts and the process set can be dynamically shrunk or grown.

## 2.2 RMA Programming Fundamentals

The following section gives an overview of the main RMA programming aspects present in MPI RMA, SHMEM, and GASPI, namely the memory management and the memory models, the communication primitives, and the completion and synchronization primitives. Since all RMA models are similar in their memory models and primitives, the concepts presented here also apply to many other RMA and PGAS programming models. The information in this section is taken from the specifications of MPI RMA [28, §12], SHMEM [19], and GASPI [27].

### 2.2.1 Memory Management and Models

The first step before any communication in RMA is the allocation or exposure of memory regions that can subsequently be accessed by remote memory accesses from other processes. To achieve that, the RMA models define functions to allocate a memory region of a given size (e.g., *MPI\_Win\_allocate*, *shmem\_malloc*) or to expose existing memory regions (e.g., *MPI\_Win\_create*). In MPI RMA and SHMEM, those calls are always called collectively by all processes, while in GASPI, memory regions can also be allocated individually.

There are subtle differences in the way how remote memory regions are represented in the three RMA models. MPI RMA represents each region in an opaque object named *window*, similar to an MPI communicator. It defines four ways of initializing windows that are all called collectively on a given communicator: (1) Using *MPI\_Win\_create*, an existing local memory region can be exposed to be remotely accessible. (2) If the MPI implementation should instead allocate the memory, then *MPI\_Win\_allocate* might be used. (3) If a window should be dynamically extensible or reducible by memory regions later on, MPI RMA allows the initialization of a so-called *dynamic window* with *MPI\_Win\_create\_dynamic*. (4) In the case that MPI processes run on the same node, windows may be allocated with shared memory using *MPI\_Win\_allocate\_shared*. Then, remote processes can perform direct load and store accesses to the window memory without going through the MPI layer. In general, the window initialization functions allow each process to pass a different size of the local window. A process can even expose no memory by passing a local window size of 0.

In SHMEM, the remotely accessible memory regions are called *symmetric data* where plain memory addresses are used in the RMA primitives to access the memory of other processes. All global and static C/C++ variables are implicit symmetric data objects. Further, SHMEM provides memory management routines such as *shmem\_malloc* that allocate memory regions on the *symmetric heap*. Memory allocations on the symmetric heap are always called collectively by all processes and are identical in size on all processes. The allocated memory regions may be located at different local memory addresses for different processes. Since the RMA primitives expect plain memory addresses to access remote memory regions, the SHMEM runtime correspondingly translates the passed memory addresses to the matching local memory address at the remote. Similar to the concept of shared memory windows in MPI RMA, the routine *shmem\_ptr* allows a process to query the local memory address of a memory region from a remote process. If both processes are located on the same compute node, the process can use this address to directly perform load/store accesses to the memory location at a remote process.

GASPI represents remote memory regions in a so-called *segment*. Each segment is allocated via *gaspi\_segment\_alloc* and has a locally unique ID per process that can be used to access a corresponding segment from a process. Before any other process can access a segment of another process, the process that allocated its own segment has to make it explicitly available using *gaspi\_segment\_register*. Alternatively, there is a collective variant *gaspi\_segment\_create* that allocates a segment at all processes and registers each

## 2 Data Races in RMA Programs

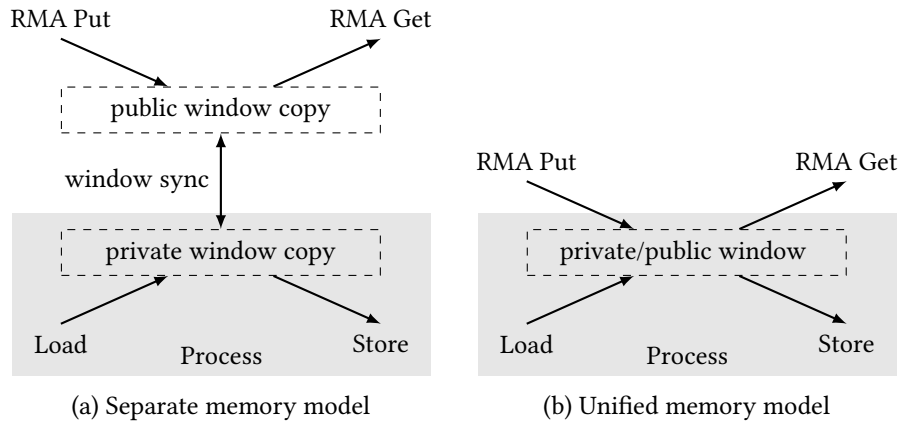


Figure 2.3: MPI RMA memory models, adapted from [38]. While SHMEM and GASPI only implement an equivalent to the unified memory model, MPI RMA also specifies a separate memory model where the user manually has to ensure coherence with additional MPI routines.

segment to be accessible from all other participating processes. In GASPI, *all* memory accesses are expected to be done through GASPI communication routines where the segment ID identifies the corresponding memory region. In case a process wants to access its own local segment with plain load/store accesses instead of using the GASPI communication routines, the routine *gaspi\_segment\_ptr* returns a local pointer to the corresponding memory region.

MPI RMA is the only approach that supports two different kinds of memory models, namely the separate and the unified memory model, as illustrated in Figure 2.3. In the *separate memory model*, the assumption is that an MPI window consists of a private and a public window copy. All local memory accesses go to the private window copy, whereas all remote accesses go to the public window copy. Modifications of the private and the public window copy have to be explicitly synchronized using the routine *MPI\_Win\_sync*. The separate memory model is intended for machines that have fast private buffers (caches) that are non-coherent to the public copy of the same memory location. In the *unified memory model*, both private and public window copy are identical, so the hardware itself ensures coherence. In today’s machines, the RDMA interconnects ensure this hardware coherence [38]. Although the separate memory model is more portable, manual coherence is typically not required in today’s machines. Therefore, the relaxed constraints of the unified memory lower the burden on the developer. SHMEM and GASPI also assume a memory model identical to the unified memory model in MPI RMA. Since the separate memory model is not of practical relevance in today’s systems and is not even present in SHMEM and GASPI, this thesis assumes that the unified memory model is used in MPI RMA. However, an extension of the race detection concepts to the separate memory model will be discussed later in Chapter 4.

## 2.2.2 Communication Routines

Any data access in RMA is done through *communication routines* that read from or write to a given remote memory location. In the following, the process that invokes the communication routine is called *origin*, while the process whose memory is accessed is called *target*. In the case of a process accessing its own memory location using a communication routine, the process is both origin and target at the same time.

The RMA models distinguish two basic RMA routines: In a *remote read* (get) call, the origin performs a read from a remote memory location at the target and writes the result to a user-specified local buffer. In a *remote write* (put) call, the origin first reads data from a user-specified local buffer and writes it to the specified remote memory location at the target. In addition, all RMA models define atomic read and write variants of those RMA routines that avoid data races with undefined behavior in case of concurrent accesses to the same memory location.

MPI RMA provides the RMA routines *MPI\_Put* and *MPI\_Get*. These routines are non-blocking, i.e., on return, the underlying put or get operation is not guaranteed to be finished. To ensure completion of these operations, designated routines have to be called, which are discussed in detail in Section 2.2.3. MPI RMA defines several additional communication routines with atomicity guarantees, namely *MPI\_Accumulate*, *MPI\_Get\_accumulate*, *MPI\_Compare\_and\_swap* and *MPI\_Fetch\_and\_op*. They atomically update, write, or read a remote memory location.

In SHMEM, the routines *shmem\_put* and *shmem\_get* provide basic RMA functionality. By contrast to MPI RMA, they are, however, *locally blocking*, i.e., they block until the user-specified local buffer is safe to be reused subsequently. SHMEM additionally defines non-blocking variants *shmem\_put\_nbi* and *shmem\_get\_nbi* that are the semantic equivalent to MPI's put and get routines. For atomicity, SHMEM defines variants of *shmem\_atomic\_fetch* and *shmem\_atomic\_set* routines that read and update single values of remote memory with atomicity guarantees.

GASPI defines the non-blocking routines *gaspi\_write* and *gaspi\_read* which are semantically equivalent to *MPI\_Put* and *MPI\_Get*. As MPI RMA and SHMEM, GASPI defines atomic operations *gaspi\_atomic\_fetch\_add* and *gaspi\_atomic\_compare\_swap* that ensure exclusivity in case of concurrent accesses to the same memory location.

## 2.2.3 Completion and Synchronization Routines

Since the RMA communication routines are one-sided, the user has to ensure completion and synchronization explicitly through additional primitives. The RMA models distinguish two completion states of an RMA operation: *Local completion* means that the memory access to the user-specified local buffer is finished such that the local buffer can be accessed safely subsequently without any side effects. For example, when a *put* operation is locally completed, the local buffer that contains the data to be written can

## 2 Data Races in RMA Programs

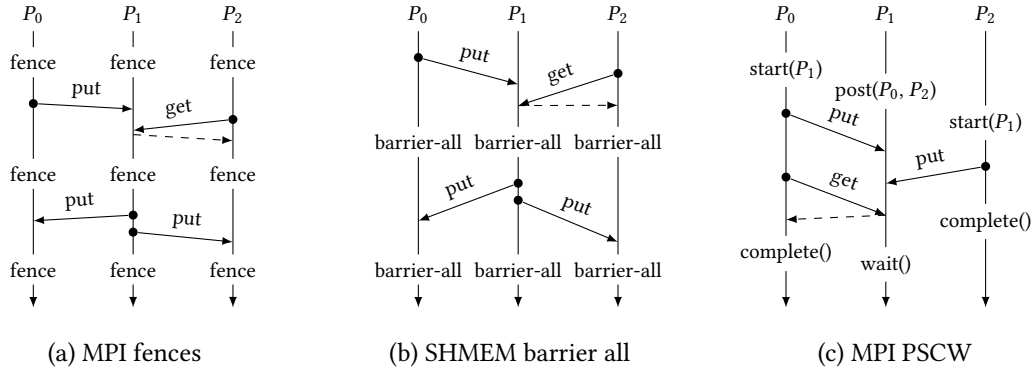


Figure 2.4: The active target completion modes in RMA specified by MPI RMA and SHMEM.

The *MPI\_Win\_fence* calls open and close access and exposure epochs. With every collective call to *MPI\_Win\_fence*, all previously issued RMA operations are guaranteed to be completed locally and remotely. The *shmem\_barrier\_all* call is the equivalent to *MPI\_Win\_fence* with the difference that there is no concept of epochs. Fine-grained active completion can be achieved via post-start-complete-wait (PSCW) in MPI RMA.

be safely reused. *Remote completion* means that the memory access is guaranteed to be visible at the target. For example, when a *put* operation is remotely completed, then the remote write is guaranteed to be visible at the target.

The following sections classify the completion and synchronization mechanisms in the three RMA models into *active target completion* where both origin and target are actively involved in the completion, *passive target completion* where completion is only ensured at the origin, and finally *signals* and *notifications* where completion is implicitly provided by polling at the target for notifications.

### Active Target Completion

The simplest way to ensure completion in RMA is bulk-synchronous active target completion, as it uses a collective call to finish any outstanding RMA operation. In MPI RMA, this kind of completion is provided by *MPI\_Win\_fence*. It is a collective call that ensures local and remote completion of any RMA operation issued before the call. More precisely, MPI RMA uses the terminology of an *access epoch* that defines the execution phase in which RMA accesses to a target might be performed at the origin and an *exposure epoch* that defines the execution phase in which an RMA access from an origin might occur at the target. A call to *MPI\_Win\_fence* opens an access epoch at the origin and simultaneously an exposure epoch at the target. If there was a previously opened access and exposure epoch, it closes the preceding access and exposure epochs before opening the new epochs.

As shown in Figure 2.4a, after the first *fence* call, any process can access the memory of all other processes, followed by another *fence* which ensures that all previously issued RMA operations are locally and remotely completed. The call to *MPI\_Win\_fence* usually

provides a barrier synchronization, but only if an access epoch or exposure epoch is closed, i.e., in particular, the first call to *MPI\_Win\_fence* is not necessarily synchronizing [28, §12.5.1]. In general, using MPI RMA fences allows for bulk-synchronous communication patterns where computation phases and communication phases alternate.

SHMEM provides with *shmem\_barrier\_all* a similar bulk-synchronous completion mechanism: It provides barrier synchronization together with the guarantee that all previously issued RMA operations are locally and remotely completed. Contrary to *fence* synchronization in MPI RMA, there is no concept of access epochs or exposure epochs in SHMEM. RMA communication calls in SHMEM can be immediately performed without any prior required call to *shmem\_barrier\_all* or similar, as also shown in Figure 2.4b. GASPI, in general, has no concept of active target completion at all.

For a more fine-grained active target completion, MPI RMA provides another mechanism called *post-start-complete-wait (PSCW)*. It avoids the global synchronization between all processes implied by a *MPI\_Win\_fence* when only certain groups of processes have to exchange data. A call to *MPI\_Win\_start* opens an access epoch to a given group of MPI processes. The origin may access the window of the given processes and finish the access epoch with a call to *MPI\_Win\_complete*. The call to *MPI\_Win\_complete* blocks until local completion of all RMA operations issued in the access epoch is ensured. On the target side, a matching call to *MPI\_Win\_post* opens an exposure epoch for the given origin processes and a call to *MPI\_Win\_wait* blocks until all RMA operations are remotely completed at the target. In Figure 2.4c, P0 and P2 open an access epoch to P1 with *MPI\_Win\_start*, while P1 opens a matching exposure epoch for P0 and P2. The calls to *MPI\_Win\_start* and *MPI\_Win\_post* do not have to block, i.e., neither *MPI\_Win\_start* has to wait for a matching *MPI\_Win\_post* to be called at the target, nor vice versa. The only synchronization between origin and target implied is that *MPI\_Win\_wait* at the target has to block until all matching *MPI\_Win\_complete* routines at the corresponding origins have been called.

### Passive Target Completion

The previously described active target completion still requires the target to actively participate in the completion of RMA operations. For truly one-sided communication, the RMA models provide routines for ensuring completion on the origin side without the target's involvement.

MPI RMA specifies a *locking* mechanism that ensures synchronized access to a target window. As shown in Figure 2.5a, a process opens an access epoch to a specific process with *MPI\_Win\_lock* and closes an access epoch with *MPI\_Win\_unlock*. When returning from *MPI\_Win\_unlock*, all previously issued RMA operations to the window at the given target are guaranteed to be locally and remotely completed. The acquired lock might be *exclusive*, which ensures that any other lock-protected access from another process to the same window at the same target may not be performed simultaneously. A process might

## 2 Data Races in RMA Programs

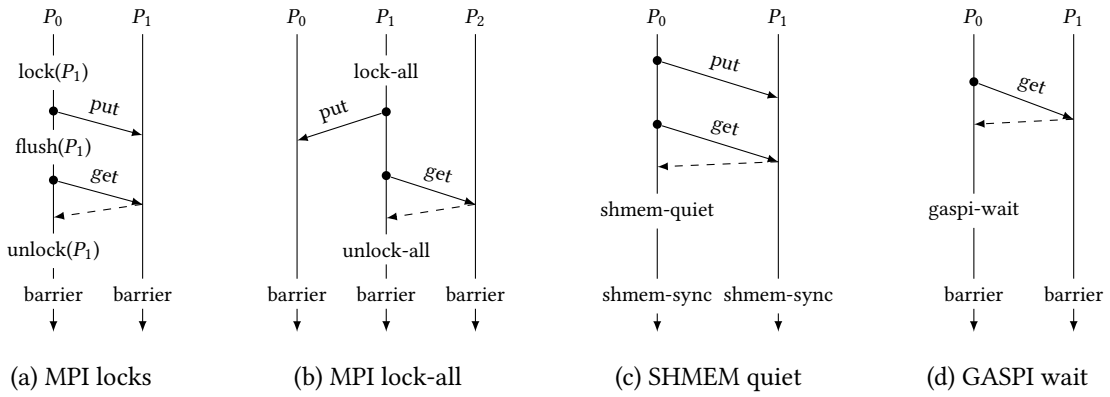


Figure 2.5: The passive target completion modes in RMA specified by MPI RMA, SHMEM, and GASPI. MPI RMA defines a locking mechanism, while SHMEM provides the call *shmem\_quiet*. GASPI specifies the call *gaspi\_wait* that only provides local completion.

also exclusively lock its own local window to perform local load and store operations without interfering with remote accesses from other processes. The lock might also be *shared* such that any other accesses protected by an exclusive lock will not be concurrent, but there is no restriction on other accesses also holding a *shared* lock. If only completion is required without closing the access epoch, then a call to *MPI\_Win\_flush* may be used. This is useful if a first set of RMA operations should be performed and completed before a second set of RMA operations. There is also a weaker variant *MPI\_Win\_flush\_local* which only ensures local completion of previously issued RMA operations.

In case a process wants to access the same window on different targets, *MPI\_Win\_lock\_all* might be used, see Figure 2.5b. It acquires a shared lock on a given window for all processes, which allows the calling process to read or modify the window memory of any process without having to lock the window of each process individually. The window can be unlocked with *MPI\_Win\_unlock\_all*. Those routines are non-collective.

Although the naming of the locking mechanism in MPI RMA may imply that a call to *MPI\_Win\_lock* blocks until a lock is acquired, this is not mandated by the MPI standard. Instead, the MPI implementation is allowed to delay the acquirement of the lock until a call to *MPI\_Win\_flush* or even *MPI\_Win\_unlock*, if there is no *flush* call in between. If the MPI implementation delays the acquisition of the lock, then it also delays corresponding RMA operations accordingly. The only exception to delayed locking is a process that wants to lock its own local window. In this case, *MPI\_Win\_lock* definitely blocks until the lock is acquired. The reason is that unlike RMA operations, local load and store accesses (performed to the own local window) are outside the MPI implementation's control and cannot be delayed.

Compared to MPI RMA, SHMEM defines a rather simple mechanism for passive target completion with the *shmem\_quiet* routine, see Figure 2.5c. It is a non-collective call that provides local completion and remote completion of all outstanding RMA operations

issued by the origin. Again, this is particularly useful if some RMA operations issued at the origin have to be completed before the next set of RMA operations.

In GASPI, there is *gaspi\_wait* that only provides local completion of all previously issued RMA operations. Since local completion of a *get* operation implies its remote completion, *gaspi\_wait* can also be used to ensure remote completion in that case. For a *put* operation, there is no completion method in GASPI to ensure its completion on the origin side. Instead, GASPI solely relies on a notification mechanism where the target waits for a signal from the origin to ensure remote completion. Details on that are explained in the next section.

The passive target completion mechanisms ensure local and remote completion at the origin side. Nevertheless, at some point in the execution, the target also has to know about the remote completion, e.g., if it wants to access the data that has been written from the origin process. For that, passive target completion calls are typically combined with process synchronization calls. The MPI examples in Figure 2.5 therefore use a call to an *MPI\_Barrier* after the *unlock* calls, the SHMEM example in Figure 2.5c uses *shmem\_sync* which is the semantic equivalent to an *MPI\_Barrier*, and the GASPI example in Figure 2.5d uses a *gaspi\_barrier*. When the processes return from the collective synchronization, the target process can safely access the previously remotely accessed memory regions.

## Signals and Notifications

A third way to ensure the visibility of remote accesses in RMA models is provided by signaling mechanisms. Here, the target explicitly waits for remote completion using a separate signaling operation. In SHMEM, the target can block in the call *shmem\_wait\_until* to wait for a remote update of a memory location by the origin. The remote update itself has to be an atomic RMA operation. Figure 2.6a shows such an example where P0 atomically sets a flag that P1 waits for. Together with a call to *shmem\_fence* (which has an entirely different meaning than *MPI\_Win\_fence*), this ensures remote completion of previously issued RMA operation at the target: A call to *shmem\_fence* provides an ordering guarantee of RMA writes from the same origin to the same target: All remote write operations issued before *shmem\_fence* will be guaranteed to be visible at the target before the remote write operations issued after *shmem\_fence*. The call itself does not provide any completion but only ordering guarantees. In Figure 2.6a, the *put* operation is guaranteed to be completed from the target perspective when *shmem\_wait\_until* returns because the *put* operation being before the fence is guaranteed to be visible at the target before the *atomic\_set* call after the fence.

SHMEM also provides a signaling mechanism coupling a plain RMA operation with a signal operation that updates a flag at the target when the RMA operation is remotely completed, as shown in Figure 2.6b with *shmem\_put\_signal*. The target itself can use a call to *shmem\_wait\_until* on the flag to wait for the signal. When this call returns, the effect of the *put* operation is guaranteed to be visible at the target. A significant difference

## 2 Data Races in RMA Programs

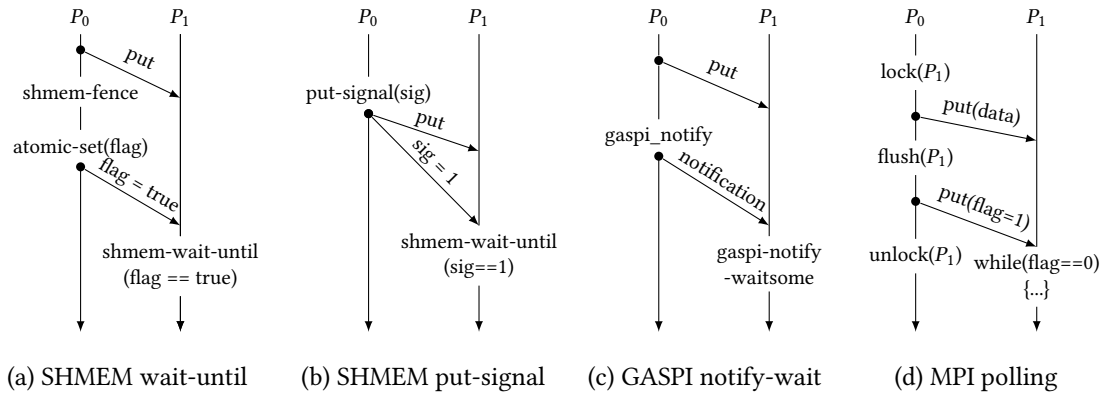


Figure 2.6: Signal and notification mechanisms in RMA.

to the previously discussed fence-wait-until mechanism is that the *shmem\_put\_signal* operation provides no guarantees regarding any other previously issued RMA operations, i.e., any other RMA operation issued before *shmem\_put\_signal* is not guaranteed to be visible on return of *shmem\_wait\_until* at the target.

The signaling mechanism in GASPI is similar to that of SHMEM and is the only way of ensuring remote completion at the target. A call to *gaspi\_notify* at the origin is matched with a call to *gaspi\_notify\_waitsome* at the target. Each notification in GASPI has an identifier such that *gaspi\_notify\_waitsome* can wait for multiple notifications if required. When the notification of an origin arrives at the target, GASPI guarantees that all previously issued RMA calls from the same origin to that target are completed. In the example of Figure 2.6c, the *put* operation from P0 is guaranteed to be visible at the target P1 when *gaspi\_notify\_waitsome* returns.

In MPI RMA, a mechanism similar to SHMEM’s signaling or GASPI’s notification is not available. However, a similar goal of notifying the target process of completion can be achieved with polling, as shown in Figure 2.6d: P0 first performs a *put* operation and flushes to ensure remote completion at the origin side. To notify the target P1 about the completion, P0 performs another *put* operation that sets a remote memory location *flag* at P1. The target process continuously polls on the memory location *flag* to wait for the flag to become true. When the target process leaves the while loop, the *put* operation with the data is guaranteed to be visible. The polling on *flag* in the while loop has to ensure MPI progress within the loop body to eventually observe the update.

## 2.3 RMA Race Classification

The one-sided nature of RMA operations and their involved completion semantics makes writing correct RMA programs difficult. Since in RMA models, processes can directly modify the memory of other processes, *data races* may occur when concurrent conflicting accesses to the same memory location at the same process are improperly synchronized.

Depending on the RMA model and the kind of conflict, a data race in RMA programming leads to undefined behavior, i.e., the outcome of the whole program is undefined, or at least to undefined results at the affected memory location. The non-deterministic nature of such races makes them often difficult to find manually.

The following subsections give an overview of the kinds of races that may occur in MPI RMA, SHMEM, and GASPI. In particular, it presents the two different race classes of *local buffer races* and *remote races* that I have introduced in [90]. I further elaborate on the extensions regarding incorrect atomicity and hybrid races that I discussed in [91].

### 2.3.1 Data Race Terminology

The term “data race” is heavily overloaded in the literature and often used interchangeably with the term “race condition”. Both, however, refer to different effects of concurrent memory accesses in parallel programs. This thesis follows the characterization of data races and race conditions done by Netzer and Miller [67]: A *data race* is a situation in which at least two memory accesses to the same memory location — with at least one being a write access — are concurrent and not properly synchronized. In other words, the exclusivity of the accesses is not ensured through any appropriate synchronization construct. A *race condition* is a situation where proper synchronization between the accesses is ensured, but the execution order of the memory accesses remains non-deterministic. This can be the case if an access to a shared variable is performed in a critical section by multiple threads, but it is still not clear which access will be performed first because it depends on which thread enters the critical section first. Netzer and Miller [67] subsume both, *data race* and *race conditions*, in the set of *general races*.

A data race is nearly always a failure and often characterized as undefined behavior by programming standards such as C++ [65] and the RMA models covered here. On the other hand, a race condition refers to the semantic property of having non-determinism in a program execution, which is not necessarily harmful. This thesis exclusively focuses on data races, which are clearly mistakes in programming that should be avoided. Whenever the term “data race” is used, it always refers to the previously mentioned definition.

### 2.3.2 Local Buffer Races

The majority of the RMA communication routines expect a user-specified local buffer that is accessed at the origin. In the case of a *get* routine, the result of the operation is *written* back to a user-specified local buffer. In the case of a *put* routine, the data that is transmitted to the target is *read* from a user-specified local buffer. Those accesses are called *buffer write* and *buffer read*, respectively.

Since most RMA communication routines are non-blocking, *writing* to the buffer of a *put* operation before the operation is locally completed may lead to undefined results

## 2 Data Races in RMA Programs

Table 2.1: Compatibility of load/store and local buffer operations to the same memory location [91].

|              | Load | Store | Buffer Read | Buffer Write |
|--------------|------|-------|-------------|--------------|
| Load         | –    | –     | ✓           | ✗            |
| Store        | –    | –     | ✗           | ✗            |
| Buffer Read  | ✓    | ✗     | ✓           | ✗            |
| Buffer Write | ✗    | ✗     | ✗           | ✗            |

```

1 int localbuf;
2 MPI_Win_fence(win);
3 if (rank == 0) {
4   MPI_Get(&localbuf, 1, MPI_INT, ..., win);
5   printf("localbuf is %d\n", localbuf);
6 }
7 MPI_Win_fence(win); // local completion

```

(a) Buffer write and load in MPI RMA

```

1 int localbuf = 0;
2 MPI_Win_fence(win);
3 if (rank == 0) {
4   MPI_Get(&localbuf, 1, MPI_INT, ..., win);
5   MPI_Put(&localbuf, 1, MPI_INT, ..., win);
6 }
7 MPI_Win_fence(win); // local completion

```

(b) Buffer write and buffer read in MPI RMA

```

1 int localbuf;
2 shmem_barrier_all();
3 if (my_pe == 0) {
4   shmem_int_get_nbi(&localbuf, &remote,
5                   1, 1);
6
7   printf("localbuf is %d\n", localbuf);
8 }
9 shmem_barrier_all(); // local completion

```

(c) Buffer write and load in SHMEM

```

1 int localbuf;
2 gaspi_barrier(GASPI_GROUP_ALL, ...);
3 if (rank == 0) {
4   gaspi_read(..., loc_seg_id, 0, ...);
5   printf("localbuf is %d\n", localbuf);
6   // local completion
7   gaspi_wait(queue_id, GASPI_BLOCK);
8 }
9 gaspi_barrier(GASPI_GROUP_ALL, ...);

```

(d) Buffer write and load in GASPI

Figure 2.7: Local buffer race examples with concurrent non-blocking RMA operations and local memory accesses in MPI RMA, OpenSHMEM, and GASPI, adapted from [91].

and for a *get* operation, *both reading and writing* of the buffer may lead to undefined results. The reason is that the RMA implementation has the freedom to access the local buffer at any point in time between the call to the RMA routine and the call to a local completion routine, so the user should not interfere with those accesses. Such situations of concurrent accesses with buffer reads or writes are called *local buffer races*. Table 2.1 gives an overview of which buffer read and write operations conflict with each other and local load and store accesses. The RMA models address the undefined outcome of local buffer races explicitly: MPI RMA [28, §12.3, p.566] states that those conflicting accesses “should not” be done, SHMEM [19, §4.2, p.8] defines it as “undefined behavior”, and GASPI [27, §8.2, p.61, p.63] states that such concurrent accesses may lead to an “undefined interleaving” of data.

Figure 2.7 shows different examples of local buffer races: Figure 2.7a shows a *get* operation where the *localbuf* variable is read in the *printf* function *before* completion is ensured with the call to *MPI\_Win\_fence*. Similarly, there are two RMA operations in Figure 2.7b that concurrently use the same buffer, leading to a local buffer race. Figure 2.7c and Figure 2.7d show similar examples in SHMEM and GASPI.

Table 2.2: Compatibility of RMA operations and local load/store accesses to the same memory location at a target process [91].

|                     | Local Load | Local Store | Remote Read | Remote Write | Remote Atomic Read | Remote Atomic Write |
|---------------------|------------|-------------|-------------|--------------|--------------------|---------------------|
| Local Load          | –          | –           | ✓           | ✗            | ✓                  | ✗                   |
| Local Store         | –          | –           | ✗           | ✗            | ✗                  | ✗                   |
| Remote Read         | ✓          | ✗           | ✓           | ✗            | ✓                  | ✗                   |
| Remote Write        | ✗          | ✗           | ✗           | ✗            | ✗                  | ✗                   |
| Remote Atomic Read  | ✓          | ✗           | ✓           | ✗            | ✓                  | ✓*                  |
| Remote Atomic Write | ✗          | ✗           | ✗           | ✗            | ✓*                 | ✓*                  |

\*when adhering to atomicity semantics in the RMA programming model

The problem of local buffer races is not limited to RMA models but is present in any programming model providing non-blocking operations with buffer accesses. In MPI non-blocking point-to-point communication, for example, the user-specified buffer should also not be accessed in a conflicting way to avoid undefined results [28, §3.7.2, p.78].

### 2.3.3 Remote Races

The remote memory access at the target associated with an RMA communication routine has to be synchronized properly with (1) other remote accesses and (2) local memory accesses of the target itself. For that, the RMA models provide different completion routines as described in Section 2.2.3. In the first step, remote completion at the origin side has to be ensured with an appropriate RMA completion routine. In the second step, the target has to be made aware that the RMA operation is completed. Depending on the chosen completion variant, the first and the second step may be coupled together in a single routine (active target completion), or additional synchronization calls may be required (passive target completion). If those steps are not correctly done for two conflicting memory operations such that they are concurrently accessing the same memory location at the target, then this situation is called a *remote race*, resulting in undefined behavior.

Table 2.2 shows which memory accesses at the target to the same memory location are conflicting and which are not. The terms “local load” and “local store” refer to plain local memory accesses that the target performs itself, and the “remote” accesses refer to RMA operations issued from remote processes. Accesses that would usually be conflicting are safe if both are atomic RMA operations, but only as long as they adhere to the atomicity semantics of the RMA model, as will be discussed in Section 2.3.4 in detail. MPI RMA [28, §12.7, p.609] and SHMEM [19, §4.2, p.8] both classify concurrent conflicting accesses at a remote memory location as “undefined behavior”. GASPI does not explicitly mention the semantics of concurrent accesses, but in any case, the concurrent accesses will at least result in undefined values at the remote memory location.

## 2 Data Races in RMA Programs

| P0 (origin)                     | P1 (target)             | P0 (origin)                     | P1 (target)             |
|---------------------------------|-------------------------|---------------------------------|-------------------------|
|                                 | memory location X       |                                 | memory location X       |
| MPI_Barrier                     | MPI_Barrier             |                                 | X = 42                  |
| MPI_Win_lock(P1)                |                         | MPI_Barrier                     | MPI_Barrier             |
| buf = 42                        | ...                     | MPI_Win_lock(P1)                |                         |
| <b>MPI_Put(&amp;buf, P1, X)</b> | <b>print(X) // race</b> | <b>MPI_Get(&amp;buf, P1, X)</b> |                         |
| MPI_Win_unlock(P1)              |                         |                                 |                         |
|                                 |                         |                                 |                         |
| MPI_Barrier                     | MPI_Barrier             | MPI_Barrier                     | MPI_Barrier             |
|                                 | print(X) // X:42        | MPI_Win_unlock(P1)              | <b>X = 1337 // race</b> |

(a) Between remote write (put) and local load (print)

(b) Between remote read (get) and local store across an MPI barrier

Figure 2.8: Remote race examples in MPI RMA using locks. Conflicting statements leading to the race are bold. Adapted from [93].

Remote races may occur in various ways due to the different completion and synchronization mechanisms in RMA models. Figure 2.8a shows an example with P0 using a *put* operation to write data to memory location X at P1 while P1 itself performs a local load of X in the *print* function. The two accesses are conflicting and concurrent, leading to a remote race. If the *print* function was moved behind the second barrier, as also shown in the example, then the *put* operation would be guaranteed to be visible such that 42 would be printed. The remote race example in Figure 2.8b shows that synchronization with barriers alone is not enough to avoid remote races: Since the *get* operation is completed only with the *MPI\_Win\_unlock* called *after* the barrier, it is concurrent with the local store to X = 1337, leading to a remote race.

A remote race can also involve multiple remote processes, as shown in Figure 2.9a. P0 writes a value to memory location X at P1, while P2 concurrently reads from it. Although P1 stays completely passive in this scenario, there is still a race on its memory location X. It is also possible to have two remote accesses from the same origin that lead to a race at the target, as the example in Figure 2.9b shows. There is no guarantee which of the two *put* operations to P1 will arrive first, leading to a remote race at the memory location X of P1. A possible solution in the case of SHMEM is a *shmem\_fence* between the two operations, since it ensures that the first *put* operation gets visible before the second *put* operation, resolving the remote race. The example in Figure 2.8b also shows that contrary to shared-memory programming, races can occur even between remote accesses originating from the same process.

Figure 2.10a shows another remote race example with a *put* operation concurrent to the local load of a *print* function. The combination of a *shmem\_fence* and a *shmem\_atomic\_set* with a *shmem\_wait\_until* ensures that the second call to the *print* function is defined and returns 1337. Finally, Figure 2.10b shows an example of a *gaspi\_write* racing with a *print* call at the target. The call to *gaspi\_wait* provides only local completion but no remote completion. Thus, the *print* statement after the barrier still races with the *gaspi\_write*. The only way of ensuring visibility of remote accesses in GASPI is by using notifications, as described in Section 2.2.3.

| P0 (origin)                                  | P1 (target)                            | P2 (origin)                     | P0 (origin)                      | P1 (target)                            |
|--|--|---------------------------------|----------------------------------|--|
| shmem_barrier_all                            | memory location X<br>shmem_barrier_all | shmem_barrier_all               | shmem_barrier_all                | memory location X<br>shmem_barrier_all |
| ...  |  |                                 | buf1 = 42                        |  |
| buf = 42                                     |  |                                 | buf2 = 1337                      |  |
| <b>shmem_put(X,&amp;buf,P1)</b> // race on X |  | <b>shmem_get(&amp;buf,X,P1)</b> | <b>shmem_put(X,&amp;buf1,P1)</b> |  |
|  |  |                                 | <b>shmem_put(X,&amp;buf2,P1)</b> |  |
| shmem_barrier_all                            | shmem_barrier_all                      | shmem_barrier_all               | shmem_barrier_all                | shmem_barrier_all                      |

(a) Between remote write (put) and remote read (get) from different origin processes

(b) Between two remote writes (put) from the same origin in SHMEM

Figure 2.9: Remote race examples using SHMEM barriers. Conflicting statements leading to the race are bold. Adapted from [93].

| P0 (origin)                     | P1 (target)                     | P0 (origin)                       | P1 (target)                            |
|---------------------------------|---------------------------------|-----------------------------------|--|
| shmem_sync                      | memory location X<br>shmem_sync | gaspi_barrier                     | memory locations X, Y<br>gaspi_barrier |
| buf = 1337                      |                                 | buf = 1337                        |  |
| <b>shmem_put(X,&amp;buf,P1)</b> |                                 | <b>gaspi_write(X,&amp;buf,P1)</b> |  |
| shmem_fence                     | <b>print(X)</b> // race         | gaspi_wait                        |  |
| shmem_atomic_set(flag,1,P1)     | shmem_wait_until(flag==1)       | gaspi_barrier                     | gaspi_barrier                          |
|                                 | print(X) // X:1337              |                                   | <b>print(X)</b> // race                |
| shmem_sync                      | shmem_sync                      |                                   |  |

(a) Between remote write (put) local load (print) using SHMEM wait-until

(b) Between remote write (put) and local load (print) in GASPI

Figure 2.10: Remote race examples using SHMEM wait-until and GASPI wait. Conflicting statements leading to the race are bold. Adapted from [93].

### 2.3.4 Incorrect Atomicity

When atomic RMA operations are used, then concurrent accesses to the same memory location are exclusive in the sense that the result will be as if they were executed in some arbitrary serial order. MPI RMA [28, §12.7.1], SHMEM [19, §3.2], and GASPI [27, §10] define specific requirements that have to be met by the atomic RMA operations. First, the atomic operations have to use the same data type. For MPI RMA, the atomicity guarantees are only given on the level of the predefined data type if a derived data type is used. For example, if a vector of *MPI\_INT* is updated in an atomic RMA operation, the individual *MPI\_INT* entries of the vector are guaranteed to be atomically updated, but not the whole vector type. Second, the accesses have to be aligned correctly and should not overlap, e.g., due to different byte offsets. In the case of SHMEM, additional restrictions on atomicity guarantees apply depending on the so-called *communication context*, which groups together RMA operations in terms of ordering and completion.

If any of the previously mentioned atomicity requirements are not met, then atomicity is not guaranteed so that concurrent accesses will lead to a remote race. Figure 2.11 shows an example in MPI RMA with atomic RMA operations using different data types, leading to a remote race. The atomicity guarantees are only provided between the RMA

## 2 Data Races in RMA Programs

```
1 if (rank == 0) {
2     short value = 1; int target = 1;
3     MPI_Accumulate(&value, 1, MPI_SHORT, target, 0,
4                   1, MPI_SHORT, MPI_SUM, win);
5 }
6 if (rank == 2) {
7     int value = 2; int target = 1;
8     MPI_Accumulate(&value, 1, MPI_INT, target, 0,
9                   1, MPI_INT, MPI_SUM, win);
10 }
```

Figure 2.11: Incorrect atomicity in MPI RMA: Both processes atomically update a memory location at process 1 but use different data types (MPI\_SHORT and MPI\_INT), resulting in a remote race. Adapted from [91].

atomic operations itself. RMA atomic operations are still conflicting with non-atomic RMA operations. Further, any atomic local load or store operation is incompatible with atomic RMA operations, e.g., C or C++ atomics cannot be used together with RMA atomic operations to ensure atomicity. Lastly, atomicity is only guaranteed for the remote accesses at the target associated with an RMA routine. The local buffer access at the origin is always non-atomic.

### 2.3.5 Hybrid Races

All three RMA models support multithreading and are thread-safe such that any RMA routine may be called by any thread at any time, making it possible to use OpenMP [7] together with RMA. The RMA completion routines, however, always apply to process scope: If there are two RMA communication routines called from different threads in the same origin process, then a subsequent call to an RMA completion routine will complete *all* previously issued RMA operations on the origin, independent of the calling thread.

Multithreading adds another level of complexity to RMA programming that may lead to local buffer races and remote races. On the origin side, if one thread calls an RMA completion routine but runs concurrently with a second thread that calls an RMA communication routine, there is no guarantee that the second thread's RMA communication will be completed. Further, if one thread issues an RMA operation with a local buffer access, then another thread performing a concurrent conflicting memory access to the same memory location results in a local buffer race.

On the target side, synchronization with the origin may only be ensured with a single thread of a target, but not with the other threads of the target, potentially leading to remote races. Figure 2.12 shows such a remote race occurring in an MPI+OpenMP program. Process 0 is single-threaded and performs an *MPI\_Put* operation that is completed with *MPI\_Win\_unlock* and synchronized with process 1 using an *MPI\_Barrier*. Process 1 executes a parallel region with two threads and two tasks: The first task synchronizes with the origin using the *MPI\_Barrier*, while the second task accesses the memory location of *winbuf* that is also written from the *put* operation of the origin. Only the thread that

```

1 if (rank == 0) {
2   MPI_Win_lock(MPI_LOCK_EXCLUSIVE, 1, 0, win);
3   int value = 42; int target = 1;
4   // write to winbuf[0] at target
5   MPI_Put(&value, 1, MPI_INT, target, 0, 1, MPI_INT, win);
6   MPI_Win_unlock(1, win);
7   MPI_Barrier(MPI_COMM_WORLD);
8 }
9 if (rank == 1) {
10  #pragma omp parallel num_threads(2)
11  {
12    #pragma omp single
13    {
14      #pragma omp task
15      { MPI_Barrier(MPI_COMM_WORLD); }
16      #pragma omp task
17      { printf("winbuf is %d\n", winbuf[0]); }
18  } }

```

Figure 2.12: Remote race in MPI RMA and OpenMP: Process 0 writes to the window of process 1 which is concurrent to the local read of the second task in the parallel region of process 1. Adapted from [91].

executes the first task with the *MPI\_Barrier* synchronizes with the origin. If the second task with the call to *printf* is executed by another thread, then the missing synchronization with the origin leads to a remote race. The race could be fixed by adding synchronization between the execution of the first and the second task, e.g., using a *taskwait* construct or task dependencies. This remote race only occurs in the case of a parallelized execution with OpenMP: If the execution is single-threaded or the code is not compiled with OpenMP support at all, the race does not occur.

### 2.3.6 Benign Races

A data race in RMA is typically undefined behavior, which means that the result of the program execution could result in anything. This is, for example, also true for the memory models of C and C++, where a data race can never be harmless [8]. SHMEM also follows the C/C++ memory model and defines any data race as undefined behavior and also for GASPI at least undefined results at the affected memory location can be expected. The memory model of MPI RMA, however, allows in certain situations for concurrent accesses leading to data races [28, §12.7, p.608]: First, the target may perform a local load concurrent to a remote write (put) to the same memory location. There is no guarantee in which order the data of the remote write will be written, but once data has been updated at the memory location by a remote access, it will stay there until another remote update of the memory location is performed. This exception allows polling on memory locations for value *changes*, e.g., from zero to non-zero, as shown in Figure 2.13. Second, the target may perform a local store concurrently with a remote read (get), again without any ordering guarantees of how the data becomes visible.

## 2 Data Races in RMA Programs

| P0 (origin)                   | P1 (target)                            |
|-------------------------------|--|
|                               | memory location X (initialized with 0) |
| MPI_Barrier                   | MPI_Barrier                            |
| MPI_Win_lock(P1)              |  |
| buf = 1                       |  |
| <b>MPI_Put(&amp;buf,P1,X)</b> | while(X == 0)                          |
| MPI_Win_unlock(P1)            | noop() // MPI progress (not depicted)  |
| MPI_Barrier                   | MPI_Barrier                            |

Figure 2.13: Polling example in MPI RMA: P1 polls on the state change of memory location X which is updated from P0.

Although those described race situations could be called *benign*, they still fulfill all conditions of a data race, and an application developer should only use them with care to avoid undefined behavior. The MPI RMA polling mechanism in Figure 2.13 can instead also be implemented with atomic *put* and *get* accesses at the origin and target side. Moreover, other RMA models, such as SHMEM, explicitly disallow such concurrent accesses. In this thesis, all races are treated as harmful since they are always avoidable and even undefined behavior in some RMA models.

### 2.3.7 Related Work

The problem of data races in RMA models has also been described and classified in other works, specifically for MPI RMA: Park and Chung [74] discuss data races caused by concurrent MPI RMA communication calls and how to detect them. Compared to the classification shown here, they only consider conflicts between remote memory accesses, not the conflicts that might occur with local memory accesses. Moreover, in the context of data race detection for MPI RMA programs, Chen et al. [16] describe the error classes named “memory consistency errors within an epoch” and “memory consistency error across processes”. Other works [2, 22] also rely on that definition. Those classes correspond to the local buffer races and remote races mentioned in this thesis, respectively.

The MPI Bugs Initiative [57] classifies errors in MPI programs according to their root cause and provides a collection of synthetic code examples that represent such errors. The authors mention the class *local concurrency* representing concurrent access situations where the buffer of a non-blocking MPI operation is accessed in a conflicting way before it is finished. This also includes the local accesses done by MPI RMA operations and is equivalent to the local buffer race class discussed in this thesis. Further, the class *global concurrency* considers situations where concurrent MPI RMA accesses and local memory accesses at the target to the same memory location with at least one write access leads to a data race, equivalent to the remote race class explained in this thesis. The error classification introduced by the MPI Bugs Initiative has also been used by the RMA race detection approaches [82, 106] of the same authors.

For SHMEM, there is only one work [5] mentioning the problem of data races without classifying them any further. For GASPI, Krzikalla [49] describes different race situations occurring due to concurrent accesses, but without explicitly classifying them. The work also specifies a generalization of the completion and synchronization semantics for race detection in the three RMA models MPI RMA, SHMEM, and GASPI, also together with OpenMP.

## 2.4 Results and Discussion

This chapter introduces the terminology of a data race in RMA programs and clarifies its semantics. Due to the high variety of synchronization mechanisms in RMA programming, there are many different ways of how data races may occur. Therefore, this chapter provides a classification of races in RMA. It mainly distinguishes between local buffer races and remote races, but also considers the semantics of atomic RMA operations which are incompatible with language atomics, e.g., in C/C++, further complicating the correct usage of RMA.

The presented classification of RMA races shows that the RMA models mostly agree on the memory models and their consistency semantics, i.e., a data race in an RMA program written in MPI RMA is typically also a data race in a semantically equivalent program written in SHMEM and GASPI. The main differences are in how completion and synchronization are achieved through active completion, passive completion, and signaling mechanisms. Most concepts can be translated from one RMA model to the other, e.g., an *MPI\_Win\_fence* is semantically equivalent to a *shmem\_barrier\_all*, while there are also concepts such as *shmem\_fence* for which no equivalent construct in the other RMA models exists.

In summary, the non-deterministic behavior of data races in RMA makes their detection with tool support desirable. Especially novice users who are unaware of the completion semantics profit from tool feedback on whether their written code is free of data races. But also hidden data races that so far did not manifest as a failure in the execution might be detected with such a tool. In the following chapters, the design and implementation of an RMA race detector are discussed. This in particular requires analyzing the synchronization and consistency semantics of the RMA models.



# 3 Clock-Based Synchronization Analysis for Distributed-Memory Programs

Parallel program executions involve multiple processing entities running independent execution streams in parallel. At some point in the execution, synchronization between the parallel entities is typically required to ensure a certain execution order. In distributed-memory programming such as with MPI [28], for example, bulk-synchronization may be established with barriers, and point-to-point synchronization between processes is established between the sender and receiver of a message exchange. In remote memory access (RMA) programming, processes may also perform synchronization via resources. For example, processes may use locks or poll on a state change of a memory location updated from remote. For correctness analyses, understanding and tracking the synchronization state of processes in a parallel program execution is a fundamental building block, as it allows for two arbitrary events in a parallel execution to decide whether they are concurrent or not. In particular, many dynamic data race detectors rely on the happened-before relation [55] that may be captured by vector clocks [24, 87].

This chapter discusses a clock-based model for an on-the-fly tracking of synchronization in distributed-memory programs, which also includes the class of RMA programs. The captured synchronization includes the traditional collective and point-to-point synchronization and the synchronization implied by RMA semantics. To that end, the different synchronization patterns of the programming models MPI, SHMEM, and GASPI are first identified and categorized. Based on this categorization, this chapter presents a set of generic synchronization primitives, and a corresponding generic vector clock exchange model to record the synchronization during the program's runtime. To showcase the model's applicability in practice, the vector clock exchange has been implemented as part of the MUST correctness checker [36]. The defined and implemented vector clock exchange enables different application scenarios for concurrency analysis, such as data race detection in RMA programs. This chapter also includes an overhead evaluation on the SPEC MPI 2007 benchmarks, which shows that the slowdown due to the vector clock exchange in real-world applications is typically 1.1x to 3.5x for up to 768 processes.

This chapter is organized as follows: Section 3.1 provides initial background knowledge on the happened-before relation and vector clocks. In Section 3.2, the different synchronization concepts in MPI, SHMEM, and GASPI are classified. Section 3.3 describes

the generic vector clock exchange model for an on-the-fly tracking of synchronization. The model's implementation in the correctness checking tool MUST is described in Section 3.4 and the results of the overhead evaluation on the SPEC MPI 2007 benchmarks are presented in Section 3.5. Section 3.6 discusses an extension of the model to hybrid parallelism with OpenMP. Finally, Section 3.7 presents different use cases of the model and Section 3.8 compares the approach to related work.

The vector clock model presented in this chapter has been developed in the course of several works: I have proposed an initial clock model specifically for MPI in my master thesis [88]. Felix Tomski generalized the vector clock model to SHMEM and GASPI and significantly extended and improved the implementation in his master thesis [103] that I have supervised. The model and its evaluation have been summarized and finally published in a paper [89]. The Sections 3.2, 3.3, 3.4, and 3.5 are based on the results of this work.

## 3.1 Happened-Before Relation and Vector Clocks

The analysis of parallel program executions often requires capturing the global ordering between events, i.e., if two events from different processes are concurrent or if one event precedes the other. A naive approach is to record each event's physical time and later compare the clocks to check whether one event preceded another. However, this would require perfectly synchronized and accurate real-time clocks on all systems (which is infeasible), and it would not be clear how two events are defined as concurrent if they are always totally ordered by the physical time they occur. The happened-before order introduced by Lamport [55] solves this problem by defining a relation between events in a parallel execution and introducing the concept of *logical clocks*.

In the following, a distributed-memory system model with  $N$  single-threaded processes  $P := \{P_0, \dots, P_{N-1}\}$ , as introduced by Schwarz and Mattern [87], is assumed. The processes can communicate data via message exchanges to each other. Each process  $P_i$  performs different actions resulting in observable *events*. The set of all events occurring on a single process  $P_i$  is called  $E_i := \{e_0^i, e_1^i, e_2^i, \dots\}$  where the individual events are enumerated in the order of their occurrence. The set  $E := E_0 \cup E_1 \cup \dots \cup E_{N-1}$  represents all events that occurred during the execution. The system model assumes three different kinds of events: (1) *Internal events* are local to the executing process and not of relevance for other processes, (2) *send events* represent the sending of a message to another process, and (3) *receive events* denote the corresponding reception of a sent message from another process. Receive events are blocking because they wait until a matching message from another process has arrived. Send events do not wait for the reception at the receiving process.

The happened-before relation  $\xrightarrow{hb}$ , originally defined by Lamport [55] and later also named “causality relation” [87], is a strict partial order describing the causal relationship of events in a parallel program execution:

**Definition 3.1** (adapted from [87]). *The happened-before order  $\xrightarrow{hb} \subseteq E \times E$  is the smallest transitive relation satisfying the following conditions:*

- (1) *If  $e_j^i, e_k^i \in E_i$  occur in the same process  $P_i$  and  $j < k$ , then  $e_j^i \xrightarrow{hb} e_k^i$ .*
- (2) *If  $s \in E_i$  is a send event and  $r \in E_j$  is the corresponding receive event, then  $s \xrightarrow{hb} r$ .*

The first part of the definition states that all events in a single process are totally ordered by their enumeration. This total order of events on a single process is often called program order  $\xrightarrow{po}$ . The second part describes the synchronizing effect of the message exchange between two processes. Two events  $a, b \in E$  are called *concurrent*, denoted as  $a \parallel b$ , if neither  $a \xrightarrow{hb} b$  nor  $b \xrightarrow{hb} a$  holds.

The happened-before relation can be recorded by using *logical clocks*. With Lamport clocks [55], each event  $e \in E$  is assigned a single clock value  $C(e) \in \mathbb{N}$ . Each process maintains the clock value, which is incremented on each event and piggybacked on every message exchange. The single value of Lamport clocks is enough to fulfill the condition that for two events  $a, b \in E$ , it holds  $a \xrightarrow{hb} b \implies C(a) < C(b)$ , but not the reverse case. An extension to the Lamport clock called *vector clock* has been defined independently by Schwarz and Mattern [87] and Fidge [25]. Instead of a single value, it stores a vector of integers  $V(e) \in \mathbb{N}^N$  for each event  $e$ . The value  $V(e)[i]$  denotes the clock value of process  $i$  stored for event  $e$ . The vector clock exchange is defined as follows:

**Definition 3.2** (adapted from [87]). *Let  $P_0, \dots, P_{N-1}$  denote the processes of a distributed computation. The vector clock  $V_i$  of process  $P_i$  is maintained according to the following rules:*

- (1) *Initially,  $V_i[k] := 0$  for  $k = 0, \dots, N - 1$ .*
- (2) *On each internal event  $e$ , process  $P_i$  increments  $V_i$  as follows:  $V_i[i] := V_i[i] + 1$ .*
- (3) *On sending message  $m$ ,  $P_i$  increments  $V_i$  as in (2) and attaches the new vector to  $m$ .*
- (4) *On receiving a message  $m$  with attached vector  $V(m)$ ,  $P_i$  increments  $V_i$  as in (2) and then updates its current  $V_i$  as follows:  $V_i := \max\{V_i, V(m)\}$ .*

The max operation is defined as element-wise maximum operation on the entries, i.e.  $\max\{V_i, V_j\} := [\max\{V_i[0], V_j[0]\}, \dots, \max\{V_i[N - 1], V_j[N - 1]\}]$  for  $V_i, V_j \in \mathbb{N}^N$ .

Figure 3.1 shows how the vector clocks  $V(e_j^i)$  are annotated to the different events  $e_j^i$ . The vector clock  $V(e_j^i)$  represents the clock  $V_i$  that process  $i$  had *after* it encountered  $e_j^i$  and correspondingly updated its vector clock. The comparison operation of vector clocks is defined as an element-wise comparison of their values:

### 3 Clock-Based Synchronization Analysis for Distributed-Memory Programs

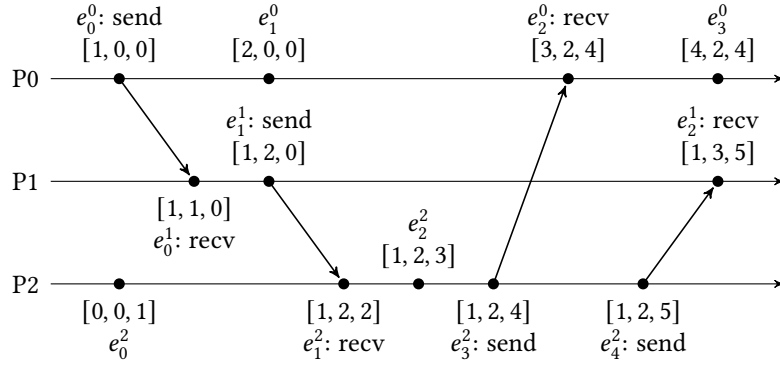


Figure 3.1: Vector clock exchange example with three processes communicating via messages. The arrows denote the message exchanges. Taken from [89].

**Definition 3.3** ([87]). *Let  $V, V'$  be vector clocks of dimension  $N$ .*

- (1)  $V \leq V'$  iff  $V[i] \leq V'[i]$  for  $i = 0, \dots, N - 1$
- (2)  $V < V'$  iff  $V \leq V'$  and  $V \neq V'$

As proven by Schwarz and Mattern [87], the comparison between vector clocks can be used to capture the happened-before order accurately:

**Theorem 3.1** ([87]). *For two events  $e$  and  $e'$  of a distributed computation, we have*

- (1)  $e \xrightarrow{hb} e'$  iff  $V(e) < V(e')$ ,
- (2)  $e || e'$  iff  $V(e) \not< V(e')$  and  $V(e') \not< V(e)$ .

Revisiting the example in Figure 3.1, it holds  $V(e_0^0) = [1, 0, 0] < [1, 1, 0] = V(e_1^1)$  implying that  $e_0^0 \xrightarrow{hb} e_1^1$ . Also, transitive synchronization effects are considered such that  $V(e_0^0) = [1, 0, 0] < [1, 2, 2] = V(e_1^2)$  implying  $e_0^0 \xrightarrow{hb} e_1^2$ , although P0 and P2 did not synchronize directly with each other up to that point. The events  $e_0^2$  and  $e_1^0$  are concurrent, since  $e_0^2 = [0, 0, 1] \not< [2, 0, 0] = e_1^0$  and  $e_1^0 = [2, 0, 0] \not< [0, 0, 1] = e_0^2$ .

The system model and the happened-before relation are designed to capture the relation of events of a *single* execution, not all possible executions. For instance, non-deterministic or out-of-order message receptions may lead to different synchronization results and, thus, different happened-before orders for different runs of the same program.

## 3.2 Synchronization in Distributed-Memory Programs

The process synchronization concepts in today's distributed-memory programs go beyond message passing assumed in the original system models for vector clock exchanges. The

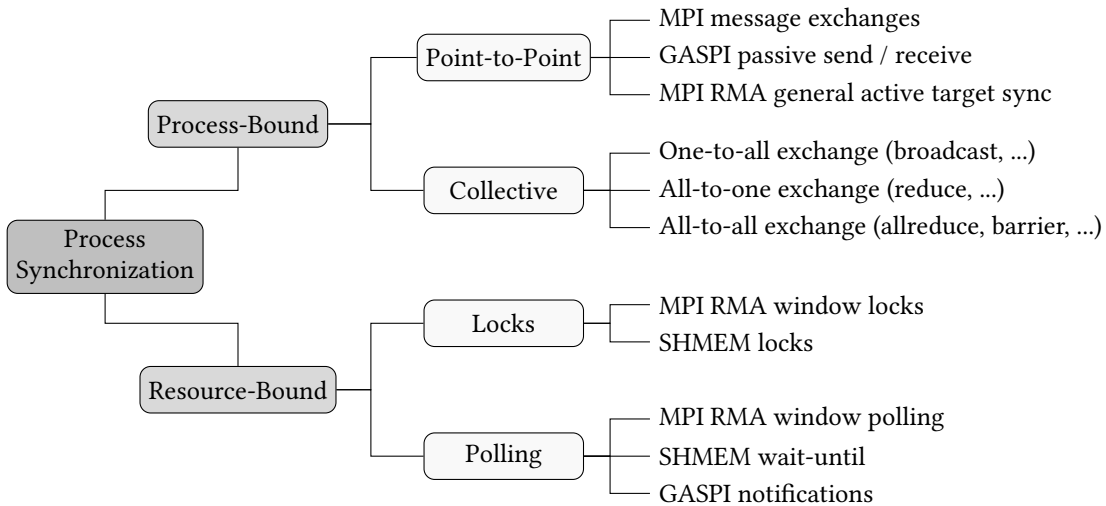


Figure 3.2: Classification of synchronization concepts present in MPI, OpenSHMEM, and GASPI. Adapted from [89].

different synchronization concepts present in MPI, OpenSHMEM, and GASPI are classified in this section.

The MPI standard characterizes the synchronization semantics of its routines by stating if they are “local” and “blocking” [28, §2.4.2], which is also used in this thesis: A procedure is *local* if returning from it does not require the invocation of a procedure on another process. Otherwise, it is called *non-local*, i.e., it may require that a procedure on another process is called. For instance, `MPI_Recv` is non-local because it requires that a matching `MPI_Send` is called, typically from another process. The procedure `MPI_Bsend` is local since it buffers the passed data to be sent out and does not require a call from any other process to return. A procedure is *blocking* if it waits for the associated operation to be completed. Otherwise, it is *non-blocking* and completion has to be ensured with a separate procedure. For instance, `MPI_Recv` is blocking since it waits until a corresponding message has been received, while its non-blocking variant `MPI_Irecv` immediately returns and requires a corresponding call to `MPI_Wait` to finish the operation. The notion of local and blocking procedures is deciding to characterize the synchronization properties of procedures in distributed-memory programming models.

The synchronization mechanisms of distributed-memory programming models can be classified into process-bound and resource-bound synchronization. *Process-bound synchronization* means that the processes synchronize “directly” with each other as they know their synchronization partners. This kind of synchronization is subdivided into point-to-point and collective synchronization. *Resource-bound synchronization* means that the processes synchronize “indirectly” through a resource, subdivided into lock and polling synchronization. Figure 3.2 shows an overview of the classes and the concrete synchronization concepts mapped to the classes that will be explained in the following.

### 3.2.1 Process-Bound Synchronization

In *process-bound synchronization*, the synchronization is employed by explicitly stating in the procedure's call arguments the processes synchronizing with each other. For instance, in a message exchange between two processes, the sender knows to which process it has sent the message, and the receiver knows from which process it received the message, at the latest after receiving the message itself. The different kinds of process-bound synchronization of MPI, OpenSHMEM, and GASPI can be subclassified as point-to-point and collective synchronization.

#### Point-to-Point Synchronization

*Point-to-point* synchronization covers all concepts that involve exactly two processes. The established synchronization may be unidirectional, i.e., one process waits for an event of the other process, or bidirectional, i.e., both processes wait for an event of each other. This synchronization behavior is particularly present in MPI point-to-point message exchanges. The usual *MPI\_Recv* call of MPI is non-local and blocks until a matching message has been received from another process. The non-blocking variant *MPI\_Irecv* is local and immediately returns, while the corresponding blocking non-local call to *MPI\_Wait* waits for the initiated receive operation to be completed. The different flavors of send procedures in MPI have different semantics: A *synchronous* send waits until a matching receive call has been issued and is therefore non-local and blocking, while a *buffered* send only stores the message in a buffer and, hence, is local and blocking.

MPI RMA provides with the general active target synchronization (PSCW) discussed in Section 2.2.3 another mechanism that implicitly provides point-to-point synchronization: The procedure *MPI\_Win\_wait* called at the target blocks until a matching *MPI\_Win\_complete* has been called at the origin.

In OpenSHMEM, there are no procedures with point-to-point synchronization semantics. Instead, OpenSHMEM provides various methods of resource-bound synchronization via locks and polling. GASPI defines the two procedures *gaspi\_passive\_send* and *gaspi\_passive\_receive* which are both blocking and non-local. The send procedure waits until the matching receive procedure has been called, so it is semantically equivalent to a synchronous *MPI\_Ssend*.

#### Collective Synchronization

In collective synchronization, processes within an arbitrary-sized group synchronize with each other. The established synchronization might be explicit, i.e., the calls are explicitly designed to provide synchronization, such as a barrier call, or implicit as a result of a data dependency. The categorization of collective operations in this chapter is similar to MPI [28, §6.2.2]:

1. In *one-to-all* synchronization, all processes wait for a single process (named root), e.g., in a broadcast operation.
2. In *all-to-one* synchronization, a single process (named root) waits for all other processes, e.g., in a reduction operation.
3. In *all-to-all* synchronization, all processes wait for each other, e.g., in a barrier or allreduce (reduce-and-broadcast) operation.

MPI provides the most extensive set of different collectives, e.g., *MPI\_Broadcast* with one-to-all synchronization, *MPI\_Reduce* with all-to-one synchronization, and *MPI\_Barrier* and *MPI\_Allreduce* with all-to-all synchronization. Similarly, OpenSHMEM defines collectives such as *shmem\_sync* with all-to-all and *shmem\_broadcast* with one-to-all synchronization. GASPI also provides *gaspi\_barrier* and *gaspi\_allreduce* as collective calls with corresponding semantics. The collective operations are local or non-local, depending on the role of the process. An *MPI\_Reduce* is non-local for the root process because it has to wait for data from all other processes but is local for all other processes.

The theoretical synchronization behavior of a collective operation might not always correspond to what is done in its concrete implementation: For instance, an *MPI\_Reduce* might be implemented as a tree-like reduction, which means that from the implementation perspective, the parent nodes in a tree would also synchronize with their children to wait for their partial results. However, this is not necessarily the case, as the non-root processes could also directly send their data to the root process and would not have to wait for any process. The data dependency (all-to-one for *MPI\_Reduce*) is the minimal amount of synchronization that is guaranteed to be established between the processes.

MPI also defines a parallel scan operation (*MPI\_Scan*) as a collective that does not fit into the three synchronization categories. Details on its synchronization behavior are discussed in Section 3.3.2.

#### 3.2.2 Resource-Bound Synchronization

Processes can also synchronize via resources instead of directly addressing processes. When using locks, a process acquires exclusive access to a specific resource. At the same time, other processes that want to acquire the lock must wait for the process to release the acquired resource, effectively synchronizing the processes. Further, with polling, a process continuously monitors and waits until a given resource (e.g., memory location) changes to a given state, also providing implicit synchronization with another process.

##### Locks

MPI RMA defines a locking mechanism for windows that coordinates concurrent access from different origins to the same target window. The target window may be locked with an exclusive lock or shared lock. When one process acquires access to a window

```
1 MPI_Win_lock(MPI_LOCK_EXCLUSIVE, rank, assert, win);  
2 MPI_Put(..., rank, ..., win);  
3 MPI_Win_unlock(rank, win);
```

Figure 3.3: Example usage of *MPI\_Win\_lock/unlock* in MPI RMA. The call to *MPI\_Win\_lock* is not required to wait to acquire the lock. With *MPI\_Win\_unlock*, the lock has to be acquired at the latest. Taken from [28, §12.5.3, p. 600].

with an exclusive lock via *MPI\_Win\_lock*, another process that also wants to lock the window has to delay its RMA operation until the other process unlocks the window with *MPI\_Win\_unlock*. Similarly, SHMEM defines a locking mechanism with *shmem\_set\_lock* and *shmem\_clear\_lock* that provides exclusive lock functionality.

It is important to note that, despite their name, calls to *MPI\_Win\_lock* do not necessarily have to block until the lock is acquired. For the example in Figure 3.3, the MPI standard allows to delay the RMA operation *MPI\_Put* and the acquirement of the lock until *MPI\_Win\_unlock* is called. In other words, the *MPI\_Win\_lock* and *MPI\_Put* operation may be buffered until the call to *MPI\_Win\_unlock* or similarly a call to *MPI\_Win\_flush*. The only case in which *MPI\_Win\_lock* blocks until the lock is acquired is when a process locks its own window. The reason is that unlike RMA operations, local load and store accesses to window memory are outside the MPI implementation’s control and cannot be delayed. Considering all those different behaviors of MPI RMA locks would significantly complicate reasoning on synchronization, especially data race detection. Therefore, the later discussed synchronization model instead overapproximates the synchronization behavior of MPI RMA locking by assuming that *MPI\_Win\_lock* always blocks until the lock is acquired. The overapproximation may lead (in exotic use cases of locks) to non-detected data races in the later discussed RMA data race detection because synchronization would be assumed to be established by *MPI\_Win\_lock* which is actually not. A detailed discussion with an example is provided in the race detection evaluation in Chapter 6.

## Polling

All three RMA models allow polling on state changes of a resource triggered by another process. In MPI RMA, a process can poll in a while loop on a local memory location to be changed from a remote process. As discussed in Section 2.3.6, MPI RMA is the only programming model that allows such polling via plain local memory accesses. In SHMEM and GASPI, this functionality is provided by explicit routines as discussed in detail in Section 2.2.3: The routine *shmem\_wait\_until* and its variants block until a given property for a given memory location is fulfilled. Similarly, GASPI provides notifications with unique identifiers. A process blocks in *gaspi\_notify\_waitsome* until another process sends a corresponding notification using *gaspi\_notify* to that process.

All the polling mechanisms have in common that a process waits for an event from another process to occur, be it an update of a memory location or a notification. Unlike

locking, polling is not about acquiring (exclusive) resource ownership but actively waiting for a state change from remote.

### 3.3 Generalized Vector Clock Exchange

The original vector clock exchange mechanism defined by Schwarz and Mattern [87] and outlined in Section 3.1 is designed to work on systems solely communicating with point-to-point messaging. However, as discussed in the previous section, there are many more ways of synchronization, such as collective and resource-bound synchronization. While collective synchronization could be translated to equivalent point-to-point messaging in terms of synchronization behavior, this is not possible in resource-bound synchronization: Since processes waiting for a resource or signaling to a resource do not know in advance with whom they are synchronizing, a more generalized notion of synchronization is required. Instead of *send* and *receive* events, the generalized model uses *signal* and *wait* events using a synchronization identifier  $\mathcal{I}$ . The adapted definition of the happened-before relation is as follows:

**Definition 3.4** (adapted from [89]). *The happened-before order  $\xrightarrow{hb} \subseteq E \times E$  is the smallest transitive relation satisfying the following conditions:*

- (1) *If  $e_j^i, e_k^i \in E_i$  occur in the same process  $P_i$  and  $j < k$ , then  $e_j^i \xrightarrow{hb} e_k^i$ .*
- (2) *If  $s \in E_i$  is a **signal event** and  $w \in E_j$  is a **matching wait event with the same synchronization identifier  $\mathcal{I}$** , then  $s \xrightarrow{hb} w$ .*

For point-to-point messaging, the original *send* event from process  $P_s$  to  $P_r$  can be represented by a *signal* event with synchronization identifier  $\mathcal{I} = (P_s, P_r)$  and the *receive* event of process  $P_r$  from  $P_s$  also correspondingly by a *wait* event with identifier  $\mathcal{I} = (P_s, P_r)$ . For resource-bound synchronization, the identifier  $\mathcal{I}$  is the corresponding resource, e.g., a memory location at a given process in case of polling or the mutex in case of locks. Collective synchronization can be modeled by a sequence of multiple *signal* and *wait* events. The modeling details will be explained in the following subsections.

The original vector clock exchange mechanism of Schwarz and Mattern [87], presented in Definition 3.2, assumes that the vector clock is piggybacked to the message. Since piggybacking is not possible in resource-bound synchronization, the vector clock is instead stored at a *deposition process*  $D(\mathcal{I}) \in P$  of the synchronization identifier  $\mathcal{I}$ . How this deposition is realized is up to the concrete implementation.

The original vector clock exchange in Definition 3.2 is generalized to work with both process-bound and resource-bound synchronization. For process-bound and resource-bound signal events, the vector clock is stored at the deposition process and retrieved and merged from it as follows:

**Definition 3.5** ([89], adapted from [87]). Let  $P_0, \dots, P_{N-1}$  denote the processes of a distributed computation. Let  $R$  be a resource. The vector clock  $V_i$  of process  $P_i$  is maintained according to the following rules:

- (1) Initially,  $V_i[k] := 0$  for  $k = 0, \dots, N - 1$ .
- (2) On each internal event  $e$ , process  $P_i$  increments  $V_i$  as follows:  $V_i[i] := V_i[i] + 1$ .
- (3) On a process-bound signal event  $s$  with  $\mathcal{I} := (P_i, P_j)$ ,  $P_i$  updates  $V_i$  as in (2) and then piggybacks its vector  $V_i$  with the event  $s$  to  $P_j$ .
- (4) On a process-bound wait event  $w$  with  $\mathcal{I} := (P_i, P_j)$ ,  $P_j$  updates  $V_j$  as in (2), waits for a matching piggybacked vector  $V(w)$  from  $P_i$ , and then updates  $V_j$  as follows:  $V_j := \max\{V_j, V(w)\}$ .
- (5) On a resource-bound signal event  $s$  with  $\mathcal{I} := R$ ,  $P_i$  updates  $V_i$  as in (2) and then merges its vector  $V_i$  with the vector  $V_R$  stored at  $D(\mathcal{I})$  as follows:  $V_R := \max\{V_R, V_i\}$ .
- (6) On a resource-bound wait event  $w$  with  $\mathcal{I} := R$ ,  $P_i$  updates  $V_i$  as in (2), fetches  $V_R$  from  $D(\mathcal{I})$ , and then updates  $V_i$  as follows:  $V_i := \max\{V_i, V_R\}$ .

In the following, the notation  $P_i \xrightarrow{s} \mathcal{I}$  means that process  $P_i$  signals to the synchronization identifier  $\mathcal{I}$ , and  $P_i \xrightarrow{w} \mathcal{I}$  means that process waits for synchronization identifier  $\mathcal{I}$ .

The distinction between process-bound and resource-bound synchronization events is needed due to slightly different clock exchange requirements. For process-bound synchronization, there is a one-to-one matching between signal and wait events, i.e., a particular signal event is matched exactly once with a corresponding wait event, so piggybacking with the events is possible. For resource-bound synchronization, multiple processes might signal to the same resource  $R$  first. Then, a single process waits on that resource  $R$ , so a single wait event might match multiple signal events and vice versa. A simple example is a shared lock: If two processes acquire a shared lock and a third process wants to gain exclusive access to that lock, it will wait for both processes to unlock, i.e., two signal events match one wait event. This is correspondingly considered in the definition of the resource-bound signal and wait event clock exchange.

The model does not restrict the concrete choice of the deposition process  $D(\mathcal{I})$  introduced with resource-bound synchronization. It could be the process where the resource is located (if applicable) or any other designated process.

The generalized vector clock exchange only assumes the minimally guaranteed synchronization provided by the programming model specification. Sometimes, the implementations of the programming models have some freedom in how to realize concepts in terms of synchronization. For example, an `MPI_Send` in MPI may be implemented as buffered send, so the sender can continue after the message has been processed locally, or as synchronous send, so the sender has to block until the message has been matched at the receiver. The latter would imply bidirectional synchronization between sender and receiver, while the former would only imply that the receiver has to wait for the

sender, but not the sender for the receiver. From the correctness perspective, assuming only the least possible synchronization is preferred as it may spot more errors in the synchronization that would otherwise remain undetected. The only exception to that is the handling of *MPI\_Win\_lock* where the synchronization is overapproximated, as already discussed in Section 3.2.2.

The following subsections provide an overview of how the concrete concepts of point-to-point synchronization, collective synchronization, locks, and polling in MPI, SHMEM, and GASPI are mapped to the generalized vector clock exchange. In the following, the terms “process-bound” and “resource-bound” are only used for signal and wait events if not implicitly clear from the context.

### 3.3.1 Point-to-Point Synchronization

Point-to-point synchronization maps naturally to process-bound signal and wait synchronization: As previously described, the sending process  $P_s$  issues a signal with  $\mathcal{I} = (P_s, P_r)$  and the receiving (waiting) process  $P_r$  issues a wait with  $\mathcal{I} = (P_s, P_r)$ . Table 3.1 shows how the different procedures in the RMA models map to signal and wait events.

MPI has many different flavors of message exchanges, especially different send modes, leading to different synchronization semantics. The buffered send *MPI\_Bsend* and ready send *MPI\_Rsend* can be mapped to a usual signal event with  $\mathcal{I} = (P_s, P_r)$ . The synchronous send *MPI\_Ssend* only returns when the receiver has posted a matching receive, so it is mapped to a signal event with  $\mathcal{I} = (P_s, P_r)$  and a wait event  $\mathcal{I} = (P_r, P_s)$  to reflect the bidirectional synchronization. Similarly, an *MPI\_Recv* is mapped to a wait event  $\mathcal{I} = (P_s, P_r)$  and if the matching message originated from an *MPI\_Ssend*, additionally to a signal event  $\mathcal{I} = (P_r, P_s)$ . Correspondingly, *MPI\_Ssend* and *MPI\_Recv* show up as both signal and wait events in Table 3.1. The plain send *MPI\_Send*, which might behave like a buffered or a synchronous send, is treated as buffered send since the model assumes the least possible synchronization, as discussed before.

For GASPI, the calls *gaspi\_passive\_send* and *gaspi\_passive\_receive* are the only calls providing point-to-point synchronization. The message send is semantically identical to a synchronous send, so the calls provide bidirectional synchronization. OpenSHMEM does not have any point-to-point synchronization primitives.

#### Handling Non-Blocking Operations

Non-blocking operations are special in the sense that they consist of two events, namely the initiation event  $e_{init}$  and a corresponding completion event  $e_{comp}$ . For example, an *MPI\_Irecv* initiates a non-blocking receive operation and is therefore  $e_{init}$ , but only finishes with a corresponding *MPI\_Wait* which would be  $e_{comp}$ . The underlying operation might take place at any point in time between  $e_{init}$  and  $e_{comp}$ . Since the generalized model

Table 3.1: Mapping of point-to-point procedures to signal and wait. Adapted from [89].

| Primitive | Procedures   |
|-----------|--|
| signal    | MPI_Send, MPI_(S B I R)send, MPI_Sendrecv, MPI_I(s b r)send, MPI_Wait(all any some) <sup>1</sup> , MPI_Test(all any some) <sup>1,2</sup> , MPI_Recv <sup>1</sup> , MPI_Probe <sup>1</sup> , MPI_Iprobe <sup>1,2</sup> , MPI_Start(all) <sup>3</sup> , MPI_Win_complete |
| wait      | MPI_Recv, MPI_Probe, MPI_Iprobe <sup>2</sup> , MPI_Ssend, MPI_Sendrecv, MPI_Wait(_all _any _some) <sup>4</sup> , MPI_Test(_all _any _some) <sup>2,4</sup> , MPI_Win_wait   |
| signal    | gaspi_passive_send, gaspi_passive_receive  |
| wait      | gaspi_passive_send, gaspi_passive_receive  |

<sup>1</sup> Only mapped to signal if this procedure is part of synchronous communication.

<sup>2</sup> Only mapped if the return flag value of the corresponding request or flag is true.

<sup>3</sup> If the corresponding persistent operation is *MPI\_Send\_init* or *MPI\_Bsend\_init*.

<sup>4</sup> Only mapped to wait if it is associated with *MPI\_Irecv*, *MPI\_Recv\_init*, or *MPI\_Issend*.

assumes that the least possible synchronization is established, the vector clock exchange uses the following rules [89]:

1. If a non-blocking operation has semantics of a signal, the signal is issued at  $e_{init}$ .
2. If a non-blocking operation has semantics of a wait, the wait is issued at  $e_{comp}$ .

For operations with signal semantics, e.g., *MPI\_Isend*, it ensures that the vector clock is immediately sent out with the call itself so that other events between *MPI\_Isend* and *MPI\_Wait* will be treated as concurrent. For operations with wait semantics, e.g., *MPI\_Irecv*, it ensures that the vector clock is received and merged only with the corresponding *MPI\_Wait*. Figure 3.4 illustrates such an exchange by the example of non-blocking MPI point-to-point communication.

For non-blocking operations in MPI, *MPI\_Wait* and *MPI\_Test* (and their variants) provide request completion. *MPI\_Wait* blocks until the underlying operation is completed, while *MPI\_Test* may be called periodically to check for request completion by checking if a flag's return value is true. *MPI\_Test* immediately returns, and only the return flag tells if the operation was completed, so the happened-before relation is only established if the return flag was true.

### Persistent Communication and Probing in MPI

Persistent communication in MPI is modeled in terms of synchronization identically to their non-persistent counterparts. The only difference is that for *MPI\_Send\_init* and *MPI\_Bsend\_init*, the signal is sent out with the *MPI\_Start* call. On the receiving side, e.g., *MPI\_Recv\_init*, the corresponding request completion call, e.g., *MPI\_Wait*, is mapped as usual to a wait event.

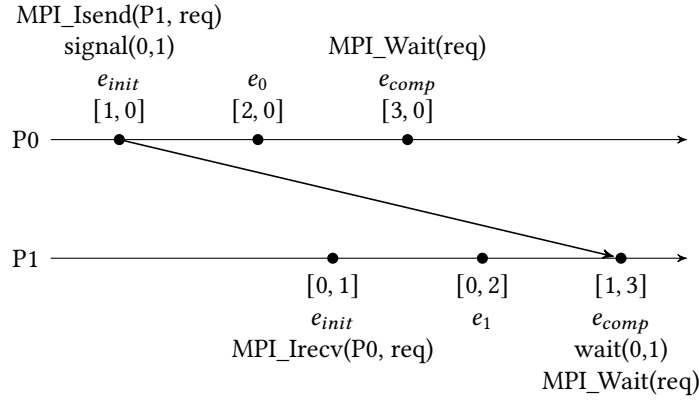


Figure 3.4: Example of handling non-blocking point-to-point operations in MPI: For *MPI\_Isend*, the signal event is mapped to the call itself, while for *MPI\_Irecv*, the wait event is mapped to the corresponding completion event triggered by *MPI\_Wait*. Using this mapping, the events  $e_0$  and  $e_1$  are considered concurrent.

*MPI\_Probe* allows to block or check for an incoming message without receiving it. From the synchronization perspective, a probe call can, therefore, be mapped to a wait event. The probe call already consumes the vector clock of the matching message such that a later called *MPI\_Recv* does not affect the vector clock and is, in that case, not mapped to any event.

### 3.3.2 Collective Synchronization

The generalized clock exchange model categorizes collective synchronization in all-to-one, one-to-all, and all-to-all primitives, as MPI does [2, §6.2.2]. The mapping of collective procedures in the RMA models is shown in Table 3.2. The all-to-one primitive represents operations where one process waits for a signal of all other processes, for example, in a reduction:

**Definition 3.6** (all-to-one [89]). *Let  $G \subseteq P$  be the group of processes participating in the operation and  $P_r \in G$  the root process. Then  $P_r \xrightarrow{w} (P_i, P_r)$  and  $P_i \xrightarrow{s} (P_i, P_r)$  for all  $P_i \in G \setminus \{P_r\}$ .*

The one-to-all primitive covers all operations where a set of processes waits for the signal of a single process as, for example, in a broadcast:

**Definition 3.7** (one-to-all [89]). *Let  $G \subseteq P$  be the group of processes participating in the operation and  $P_r \in G$  the root process. Then  $P_r \xrightarrow{s} (P_r, P_i)$  and  $P_i \xrightarrow{w} (P_r, P_i)$  for all  $P_i \in G \setminus \{P_r\}$ .*

Table 3.2: Mapping of collective procedures. Adapted from [89].

| Primitive  | Procedures   |
|------------|--|
| all-to-one | MPI_Gather(v), MPI_Reduce  |
| one-to-all | MPI_Bcast, MPI_Scatter(v)  |
| all-to-all | MPI_Barrier, MPI_Alltoall(v w), MPI_Allgather(v), MPI_Allreduce, MPI_Reduce_scatter(_block), MPI_Win_fence, MPI_Win_free   |
| one-to-all | shmem_broadcast  |
| all-to-all | shmem_barrier_all, shmem_alltoall, shmem_(f)collect, shmem_team_sync, shmem_sync_all, shmem_malloc, shmem_align, shmem_free, shmem_realloc, shmem_(and or xor max min sum prod)_reduce |
| all-to-all | gaspi_barrier, gaspi_allreduce(_user), gaspi_group_commit, gaspi_segment_(create use)  |

Lastly, the all-to-all primitive represents operations where all processes wait for all other processes as, for example, in a barrier or all-reduce:

**Definition 3.8** (all-to-all [89]). *Let  $G \subseteq P$  be the group of processes participating in the operation. Then  $P_i \xrightarrow{s} (P_i, P_j)$  and  $P_i \xrightarrow{w} (P_j, P_i)$  for all  $P_i, P_j \in G$  with  $P_i \neq P_j$ .*

This kind of abstraction does not capture the synchronization behavior of all MPI collective operations completely: As described in Section 3.2.1, the parallel scan operation provided by MPI\_Scan and MPI\_Exscan does not fit into this categorization. By definition of a parallel scan, process  $P_i$  has to receive the result of all processes  $P_j$  with  $j < i$  and thus synchronizes with them. MPI\_Scan cannot be directly mapped to a collective operation in the generalized clock exchange model, but instead is translated to point-to-point synchronization with *signal* and *wait*, i.e.,  $P_i$  waits for all processes  $P_j$  with  $j < i$  and signals to all processes  $P_k$  with  $i < k < n$ .

Another detail abstracted in the collective exchange are MPI collective operations such as MPI\_Gatherv or MPI\_Scatterv which allow specifying varying counts of exchanged elements per processor. If the root process  $P_r$  performing an MPI\_Gatherv specifies a receive count of zero elements for a certain process  $P_j$ , then  $P_r$  does not have to wait for  $P_j$  and thus does not synchronize with  $P_j$ . The model presented here does not take this specificity into account and treats those operations (e.g., MPI\_Gatherv) identical to the variants with non-varying counts (e.g., MPI\_Gather).

Figure 3.5 shows a collective vector clock exchange example with the *all-to-all* primitive. The representation of collectives with individual *signal* and *wait* events would actually mean that with an *all-to-all* primitive, a process increments its local vector clock multiple times (for all corresponding *signal* and the *wait* events). However, it is sufficient to increase the vector clock only *once* before the underlying exchange and then perform the exchange with the same clock. For example, this means for the *all-to-all* primitive that all processes have the identical vector clock after the exchange. This avoids unnecessary large vector clocks and still delivers the same information.

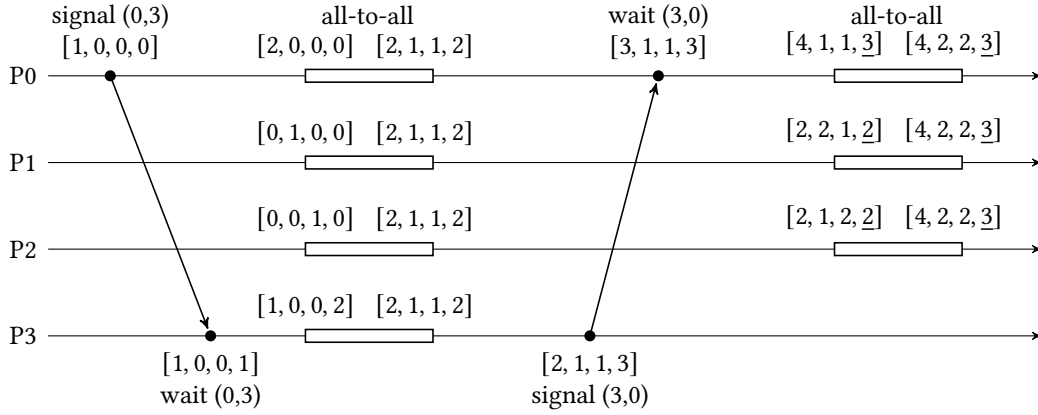


Figure 3.5: Collective vector clock exchange example with all-to-all. In the second all-to-all exchange, the processes in group  $G = \{P_0, P_1, P_2\}$  synchronize.  $P_3$  does not participate in the exchange, but still, its clock value has to be exchanged between the processes (underlined in the figure). Otherwise,  $P_1$  and  $P_2$  would miss the (transitive) synchronization with  $P_3$ . Adapted from [89] and [103].

### Optimizing Collective Exchanges

Since the collective primitives can be directly translated to several *signal* and *wait* events between a group of processes, they are syntactic sugar from a theoretical perspective and are actually not needed. However, from an implementation perspective, having information about the collective synchronization behavior allows for optimized vector clock exchanges. Such optimizations can avoid the transfer of redundant clock entries and, therefore, improve the performance of the exchange. The optimization potential depends on the collective operation primitive: The *all-to-one* primitive can be performed with an element-wise maximum reduction of the vector clocks from the different processes. Similarly, the *one-to-all* primitive can be performed with a broadcast and an element-wise combination at the receiving processes. Lastly, for the *all-to-all* primitive, any all-reduce scheme with an element-wise maximum reduction on the vector clock can be used. Internally, typical optimizations, such as tree-like implementations for the vector clock reductions, might be used. As performance analyses on real-world applications have shown [103], those collective optimizations of the vector clock exchange are elemental to avoid significant overheads.

*All-to-all* synchronization is provided in many collective operations, especially since it represents barrier synchronization between processes, so its optimization is significant for efficient clock exchanges. As discussed in [89], the local vector entry  $V_i[i]$  of process  $P_i$  must always be larger or equal to the entry  $V_j[i]$  for  $P_i$  at any other process, i.e.,

$$V_i[i] \geq V_j[i] \text{ for all } P_i, P_j \in P.$$

Instead of exchanging the full vector clock  $V_i$ , every process  $P_i$  of the collective only has to provide its entry  $V_i[i]$  so that the data exchange is an all-gather scheme with one

### 3 Clock-Based Synchronization Analysis for Distributed-Memory Programs

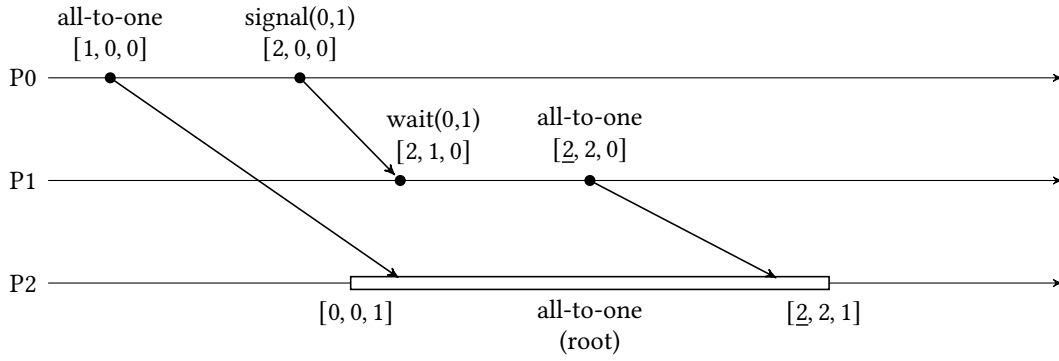


Figure 3.6: Collective vector clock exchange example with all-to-one.  $P_2$  is the root process of the all-to-one collective and waits for the clocks of all other processes. The underlined clock value is the transitive synchronization of  $P_0$  to  $P_2$  via  $P_1$ .

entry gathered per process. Figure 3.5 shows an example vector clock exchange with two *all-to-all* primitives. For the first *all-to-all* exchange, each process can send its local entry  $V_i[i]$  because all processes will have the same resulting vector clock when the operation is finished.

The discussed optimization only applies to processes of the group that participate in the collective. For other processes  $P_k \notin G$ , still all entries must be sent along to ensure that transitive synchronization with them is considered correctly. In Figure 3.5, the second all-to-all exchange works on the group  $G = \{P_0, P_1, P_2\}$ .  $P_0$  synchronizes beforehand with  $P_3$  while with the subsequent all-to-all exchange,  $P_1$  and  $P_2$  synchronized transitively with  $P_3$ . To reflect that, each process  $P_i \in G$  of the all-to-all exchange sends its local entry  $V_i[i]$  for the all-gather operation and all entries  $V_i[k]$  for the processes  $P_k \notin G$  which are then combined in a usual all-reduce maximum operation.

The all-gather optimization does not apply to the all-to-one primitive, because only the root process waits until all signals from the other processes arrive. All other processes can continue execution after sending the signal. This especially means that one signaling process  $P_i$  could encounter the all-to-one primitive, send its signal, and continue execution. Then,  $P_i$  could synchronize with another process,  $P_j$ , before  $P_j$  itself sends its signal of the all-to-one exchange. This synchronization would be missed if each process  $P_i$  only sent its local entry  $V_i[i]$ . Figure 3.6 illustrates an example where sending the local entry in an all-gather scheme is insufficient to capture synchronization with all-to-one. After  $P_0$  encounters the all-to-one primitive and sends its vector clock, it synchronizes with  $P_1$  before  $P_1$  itself encounters the all-to-one primitive. If the all-gather optimization was applied to the all-to-one exchange, the transitive synchronization of  $P_0$  with  $P_2$  would be missed. Thus, for all-to-one primitives, each signaling process has to send the whole vector clock. Further details of the collective optimizations are discussed in Felix Tomski's master thesis [103].

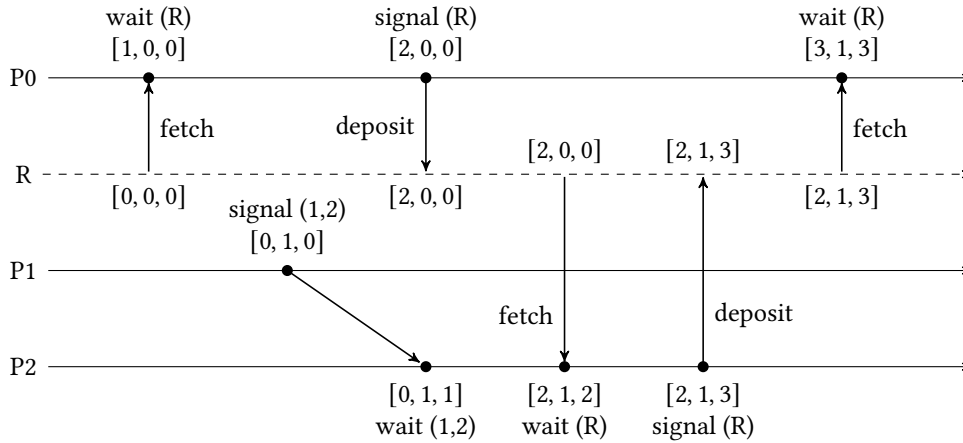


Figure 3.7: Resource-bound synchronization example using a resource  $R$ .  $P_0$  and  $P_2$  synchronize with each other using the resource  $R$  through resource-bound signal and wait events. The resource  $R$  is initialized with vector clock  $[0, 0, 0]$ . Taken from [89].

### 3.3.3 Locks and Polling

With locks and polling, synchronization between processes is established through a resource. The synchronization identifier  $\mathcal{I}$  is a resource, e.g., a mutex in case of locking or a memory location in case of polling. The deposition  $D(\mathcal{I})$  is the process where the vector clock information on the resource is stored. This process can be chosen arbitrarily, but all processes of the vector clock exchange have to agree on it.

When signaling to the resource, a process merges its current vector clock with the vector clock  $V_{\mathcal{I}}$  of the resource  $\mathcal{I}$  and stores the updated vector clock  $V_{\mathcal{I}}$  at the resource. When waiting for a resource, it fetches the vector clock  $V_{\mathcal{I}}$  from the resource and merges its own vector clock with the fetched vector clock. Any resource not previously used for synchronization is initialized with a zero vector. Figure 3.7 illustrates how this resource-bound exchange process works.

Table 3.3 shows the mapping of lock-based synchronization primitives in the RMA models present in MPI RMA and SHMEM. When a process performs a lock operation, it blocks in this operation until the process currently holding the resource (if there is any) unlocks it. Thus, a lock operation is equivalent to waiting for a resource. A process unlocking a resource is equivalent to a signal event because another process waiting for the lock has to wait for it and, therefore, synchronizes with the process that unlocked the resource. Again, as discussed in Section 3.2.2, MPI RMA locks are not necessarily implemented to always block in an *MPI\_Win\_lock* call, but the model overapproximates the implied synchronization here.

Table 3.4 lists the mappings of synchronization through polling, which has been discussed in detail in Section 2.2.3. MPI RMA permits polling on state changes of a local memory location that is updated remotely. The remote update can be any remote write access,

Table 3.3: Mapping of synchronization through locks. Adapted from [89].

| Primitive | Procedures                         |
|-----------|------------------------------------|
| signal    | MPI_Win_unlock, MPI_Win_unlock_all |
| wait      | MPI_Win_lock, MPI_Win_lock_all     |
| signal    | shmem_clear_lock                   |
| wait      | shmem_set_lock                     |

Table 3.4: Mapping of synchronization through polling. Adapted from [89].

| Primitive | Procedures / Actions  |
|-----------|---|
| signal    | (potentially) any MPI RMA write                               |
| wait      | (potentially) any local memory read                           |
| signal    | shmem_put_signal, shmem_signal_(set add), SHMEM atomic writes |
| wait      | shmem_wait_until(_any _all _some), shmem_test <sup>1</sup>    |
| signal    | gaspi_notify, gaspi_write_notify                              |
| wait      | gaspi_notify_waitsome   |

<sup>1</sup> Only mapped if the return flag value of the call is true.

modeled as a signal event, while the polling is a local memory read on the corresponding location, modeled as a wait event. SHMEM and GASPI both implement a signaling mechanism through corresponding routines that natively map to signal and wait events in the model.

In practice, it is too expensive in terms of overhead to transmit a vector clock with *any* MPI RMA write and check for synchronization with *any* local memory read. Therefore, this kind of synchronization has to be explicitly marked by the user using annotation functions, which will be discussed in Section 3.4.6.

### 3.3.4 Non-Deterministic Synchronization

The happened-before relation (and therefore the vector clock analysis) only models synchronization between events of a *particular* program execution. Thus, depending on the synchronization primitives used, the established happened-before relation might differ from run to run. Such non-deterministic synchronization may require multiple runs of an on-the-fly tool to cover all possible interleavings of executed instructions. This section briefly discusses different sources of non-determinism and how this is handled in the generalized clock exchange for some special cases.

```

1 if (rank == 0) {
2   int sendbuf = 42;
3   MPI_Send(&sendbuf, 1, MPI_INT, 1, 0,
4           MPI_COMM_WORLD);
5 } else if (rank == 1) {
6   int recvbuf; MPI_Request req;
7   MPI_Irecv(&recvbuf, 1, MPI_INT, 0, 0,
8            MPI_COMM_WORLD, &req);
9
10
11 // wait for request completion
12 MPI_Wait(&req,
13         MPI_STATUS_IGNORE);
14
15 // completed, do something
16
17 }

```

(a) *MPI\_Wait* to block on request completion.

```

1 if (rank == 0) {
2   int sendbuf = 42;
3   MPI_Send(&sendbuf, 1, MPI_INT, 1, 0,
4           MPI_COMM_WORLD);
5 } else if (rank == 1) {
6   int recvbuf; MPI_Request req;
7   MPI_Irecv(&recvbuf, 1, MPI_INT, 0, 0,
8            MPI_COMM_WORLD, &req);
9
10 int flag = 0;
11 // test once for completion
12 MPI_Test(&req, &flag,
13         MPI_STATUS_IGNORE);
14 if (flag == true) {
15   // completed, do something
16 }
17 }

```

(b) *MPI\_Test* with a onetime check for request completion.

Figure 3.8: Non-blocking receive examples in MPI.

## Sources of Non-Determinism

An obvious non-deterministic situation is a single MPI wildcard receive operation with multiple senders: The receiver only synchronizes with the sender whose message was received, there is no synchronization established with the others. The generalized vector clock exchange accordingly only captures the synchronization of the receiver with the designated sender. Another situation is the synchronization with a locking mechanism: When two processes want to lock the same resource, then the process that acquires the lock first may differ from run to run, so the established synchronization also differs from run to run. Again, the vector clock exchange only captures the established synchronization of locks for a particular run. Some examples on less obvious situations, namely non-blocking tests on completion and conditional synchronization, and their effects on the happened-before relation, are discussed in the following.

## Non-Blocking Tests on Completion

For non-blocking operations in MPI, *MPI\_Wait* and *MPI\_Test* provide request completion. As previously discussed, *MPI\_Wait* blocks until the underlying operation is completed, while *MPI\_Test* may be called periodically to check for completion through a return flag. From a synchronization point of view, *MPI\_Wait* only returns after the synchronization effect of the underlying operation is established. *MPI\_Test* immediately returns, and only the return flag tells if the operation was completed and, therefore, the happened-before relation was established.

The example in Figure 3.8 shows a send operation from  $P_0$  and a matching non-blocking receive operation from  $P_1$ . The first variant uses *MPI\_Wait* and the second variant calls *MPI\_Test* only *once*. For the second variant, the resulting happened-before relation might

|   |   |
|---|---|
| <pre> 1 // flag is already initialized to 1 2 static int flag = 1; // remotely accessible 3 if (mype == 0) { 4   // atomically sets flag at P1 to 1 5   shmem_int_atomic_set(&amp;flag, 1, 1); 6 } else if (mype == 1) { 7   // waits until flag == 1 8   shmem_wait_until(&amp;flag, 9                   SHMEM_CMP_EQ, 1); 10  // condition is immediately fulfilled 11 12 }</pre> | <pre> 1 static int flag = 0; 2 if (mype == 0) { 3   // signals to PE 2 4   shmem_int_atomic_set(&amp;flag, 1, 2); 5 } else if (mype == 1) { 6   // signals to PE 2 7   shmem_int_atomic_set(&amp;flag, 1, 2); 8 } else if (mype == 2) { 9   // waits until flag == 1 10  shmem_wait_until(&amp;flag, 11                  SHMEM_CMP_EQ, 1); 12 }</pre> |
|---|---|

(a) Wait condition is immediately fulfilled.

(b) Two processes that both set a flag to the same value.

Figure 3.9: SHMEM wait examples with ambiguous synchronization behavior.

be different from run to run: (1) It might be that the return flag of *MPI\_Test* is false, so the send from  $P_0$  did not happen before the receive operation. (2) If the return flag is true, then for the *particular* execution of the program, the send of  $P_0$  happened before the call to *MPI\_Test* at  $P_1$ . Thus, the vector clock exchange model delivers different results for this example. As previously stated, this is a general property of the happened-before analysis of event streams and not a limitation of the vector clock exchange.

Similar non-determinism can be found in the other programming models: For instance, SHMEM's call *shmem\_test*, which is the non-blocking variant of *shmem\_wait\_until* providing a return value to indicate the fulfillment of the tested condition. Thus, the vector clock model also checks the return value and assumes a corresponding *wait* event if *shmem\_test* was successful.

### Conditional Synchronization

Another closely related non-determinism in the happened-before relation results from conditioned waiting in resource-bound synchronization. The *shmem\_wait\_until* routine expects a condition to be specified as a parameter. In Figure 3.9a,  $P_1$  waits until the remotely accessible variable *flag* is set to 1.  $P_0$  sets the value of *flag* correspondingly to 1 at  $P_1$ . However, the flag is already initialized with value 1, so *shmem\_wait\_until* will run through, independently of the write access of  $P_0$ . The clock exchange model interprets the atomic write to the *flag* variable as a signal event to a resource, while the *shmem\_wait\_until* is interpreted as a wait event to the same resource. Thus, if  $P_0$  first writes its vector clock to the resource in the signal event and subsequently,  $P_1$  fetches the vector clock from the resource in the wait event, a happened-before relation is established between those events. If  $P_0$  writes to the *flag* variable *after*  $P_1$  went through *shmem\_wait\_until*, no happened-before relation is established.

Figure 3.9b shows a similar ambiguous situation with three processes.  $P_0$  and  $P_1$  both set the same value to *flag*, interpreted as *signal* events to the resource. If both  $P_0$  and  $P_1$

deposit their vector clock to the same resource before  $P_2$  gets into the *shmem\_wait\_until* operation,  $P_2$  assumes synchronization with both processes. However, in general, one process is enough to fulfill the condition, so in other executions of the same program, only one process might establish a happened-before relation with  $P_2$ .

### 3.3.5 Limitations

Due to the high complexity and variety of synchronization mechanisms in distributed-memory programming, the generic vector clock exchange model does not consider all synchronization effects and currently does not support hybrid parallelism. Those limitations will be discussed as follows.

The first limitation is the lack of support for direct translation of collective operations beyond the all-to-one, one-to-all, and all-to-all primitives. While the three primitives cover all collective synchronization methods of SHMEM and GASPI, MPI defines two collective operations with involved implicit synchronization patterns, namely *MPI\_Scan* and *MPI\_Exscan* implementing a parallel (exclusive) scan operation between processes. Both cannot be directly mapped to a collective operation in the generalized clock exchange model. However, their synchronization effects can be translated to point-to-point synchronization with *signal* and *wait*, as discussed in Section 3.3.2. Thus, advanced collective synchronization may require additional manual mapping effort.

Any form of task-based programming and synchronization has not been considered in the model, since MPI, SHMEM, and GASPI do not support them. There are only a few PGAS programming models with task (dependencies) support, such as UPC++[4] and the discontinued DASH [29] model. In terms of synchronization, such dependencies between tasks can be expressed as resource-bound synchronization. An integration of task-based programming constructs is left as future work.

The presented clock exchange model supports the three distributed-memory programming models MPI, SHMEM, and GASPI and assumes that the execution is single-threaded. All programming models support hybrid parallelism through shared-memory parallelism, especially with OpenMP [7]. The interaction of process synchronization and thread synchronization poses additional challenges for the vector clock model. Adding support for multithreading requires defining a mapping of the shared-memory synchronization concepts to the model primitives such that the synchronization between threads is captured correctly. Further, the fork-join parallelism of shared-memory programming requires handling a dynamically sized number of execution units in the vector clocks. Section 3.6 describes details on a possible extension of the model to hybrid parallelism.

## 3.4 Scalable On-the-Fly Tracking

The generic vector clock exchange model can be used to implement an on-the-fly causality tracking of events in a distributed-memory program execution. The term “on-the-fly” means that the vector clocks are maintained along with the program execution: After a relevant event  $e$  occurs in the application at process  $P_i$ , the causality tracking ensures that the vector clock  $V_i$  has been updated according to the vector clock exchange model, including merging clocks received from other processes, if required. Any subsequent (correctness) analysis relying on the vector clocks can inspect  $V_i$  on the fly after event  $e$  at  $P_i$  to determine  $V(e)$  at runtime. The vector clock exchange is implemented as distributed analysis to ensure scalability, i.e., each process  $P_i$  maintains its vector clock locally and exchanges it with other processes  $P_j$  upon synchronization.

This section discusses three essential aspects of the implementation required for such on-the-fly tracking: (1) the interception of calls to capture relevant events of the programming model, (2) the communication layer and analysis workflow to exchange the vector clocks between processes in a scalable fashion, and (3) the interaction with the user who can manually annotate synchronization that is not captured by the implementation. The vector clock analysis has been implemented in the context of Felix Tomski’s master thesis [103] as part of the correctness checking tool MUST [36] that uses the Generic Tools Infrastructure (GTI) [37]. GTI is a scalable framework for designing runtime tools for parallel programs, while MUST is a correctness tool specifically for MPI applications relying on GTI. The following subsections describe how GTI and MUST are leveraged to implement the call interception, communication layer, and user annotations for the vector clock exchange.

### 3.4.1 Call Interception

To capture the events relevant to the vector clock exchange model, the programming models’ routines must be intercepted at runtime. The profiling interfaces of MPI [28, §15.2], SHMEM [19, §10], and GASPI [27, §14.2] provide a portable way for that. For each call, e.g., *MPI\_Send*, there is a name-shifted version, e.g., *PMPI\_Send*. Tool implementors may override the original implementation of *MPI\_Send* with their own variant and may call to the original implementation using *PMPI\_Send*.

The conventional profiling interfaces only allow a single tool to override the original call. When multiple tools want to intercept the calls, the interception requires an additional layer. P<sup>N</sup>MPI [85] is an extension to the PMPI profiling interface, providing a multiplexing layer that ensures interoperability of multiple tools that all want to intercept MPI calls. The different tools are composed into a tool stack that can be adapted at runtime, e.g., to enable or disable the interception for a given tool. In the context of my research work, I

also extended<sup>1</sup> the support of P<sup>N</sup>MPI to incorporate the profiling interfaces of SHMEM and GASPI. The unified interception infrastructure enables a generic design of the vector clock exchange tracking.

The Generic Tools Infrastructure (GTI) [37] leverages P<sup>N</sup>MPI to provide a modular analysis tool framework for parallel programming models, with a focus on MPI interception. A tool developer can implement several analysis modules in GTI and specify which intercepted MPI calls should be passed to the analysis module. Such analysis modules encapsulate analysis code and algorithms, e.g., for a deadlock analysis. From P<sup>N</sup>MPI's point of view, each analysis module is a separate tool in the tool stack. Thus, analysis modules can be loaded dynamically at runtime, so only desired analyses for a particular tool run are performed. Each analysis module may perform its analysis *before* calling to the original implementation of the MPI function, also called *pre-analysis*, or *after* the call to the original implementation of the MPI function returned, also called *post-analysis*.

With the extension of P<sup>N</sup>MPI support to SHMEM and GASPI, I have also extended the support of GTI for SHMEM and GASPI. The vector clock exchange is implemented in several analysis modules, as will be discussed in detail in Section 3.4.5.

### 3.4.2 Communication Layer

In addition to intercepting relevant calls, a communication infrastructure is required that exchanges the vector clocks between processes in a scalable fashion. An analysis module in GTI can either run as *local analysis* in the application process or may be offloaded as *tool analysis*. For the latter, GTI creates additional tool threads or tool processes to reduce the computational load on the application process and manages the transfer of the relevant data from the application process to the tool thread or process.

Since for many runtime analyses, as for the vector clock tracking, information has to be exchanged between processes, GTI establishes *communication channels* between tool threads and processes. The sender of the communication channel might transmit analysis data that triggers an event on the receiver side. The receiver can then react to this event, depending on the performed analysis algorithm. The communication channels may use different underlying *communication protocols*, such as MPI point-to-point messaging, TCP, or shared memory (if possible). Further, different *communication strategies* might be used depending on whether the sender should block until the analysis at the receiver is finished or whether it may continue execution. GTI supports different blocking and non-blocking strategies for data transfer. The tool threads and processes may form a tree-based overlay network, as shown in Figure 3.10, which is especially useful for a scalable collection of analysis data in a centralized process.

<sup>1</sup>The extended infrastructure is published as part of RMA Sanitizer in <https://zenodo.org/records/12965018> and in this thesis artifact under a BSD 3-clause license.

### 3 Clock-Based Synchronization Analysis for Distributed-Memory Programs

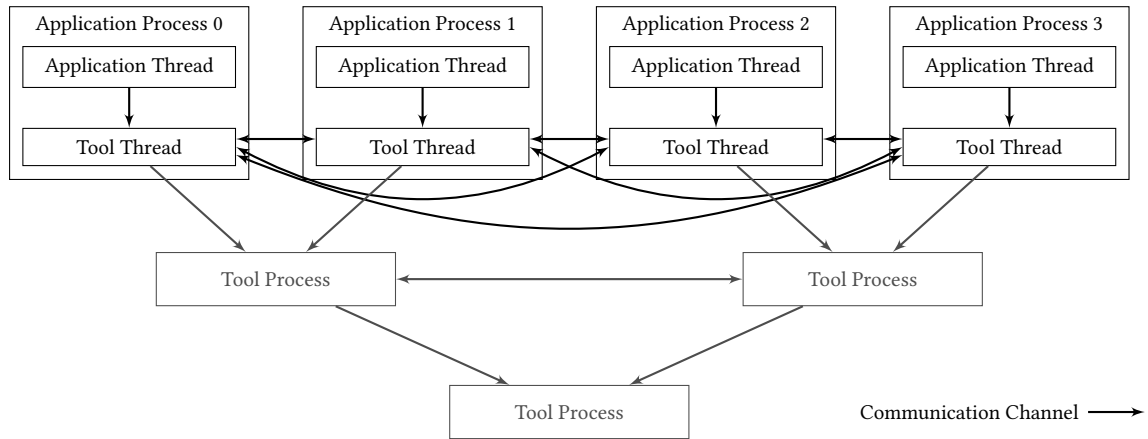


Figure 3.10: Tree-based overlay network example in GTI with four application processes: Each application process runs an additional tool thread. Additionally, there are three tool processes connected in a tree fashion to the application processes. For the vector clock exchange, only tool threads are relevant. Based on [37].

In the case of vector clock tracking, an additional tool thread is created for each application process that exchanges vector clocks with the tool threads of other application processes while the program is being executed. A GTI communication channel is established between the application thread and the tool thread and between all tool threads of the application processes. The reasoning behind that architecture is that the tool threads can process and exchange the vector clocks while the application itself can continue running independently. The application thread runs the application and whenever a relevant event occurs, it transfers that information to the tool thread. Subsequently, the tool thread performs the required analyses and exchanges (if needed) the analysis data with the tool threads of the other application processes. Reconsidering Figure 3.10, only the tool thread layer is required for the vector clock exchange, no centralized tool process or any tree-like structure is needed. Currently, the analysis implementation only supports a single application thread per process, but efforts to also support hybrid parallelized applications with multiple application threads will be discussed in Section 3.6.

The communication channel between the application thread and the tool thread uses GTI's shared memory communication protocol, while for the communication between the tool threads of the processes, a GTI protocol using MPI point-to-point messaging is chosen here. The strategy for communication between an application thread and its tool thread is blocking, i.e., when the application thread sends out analysis data to the tool thread, it waits until the tool thread has processed the data and finished the corresponding vector clock operations. This avoids the continuation of the application execution while the vector clock stored at the tool thread is outdated. Outdated vector clocks might lead to wrongly attributed events in terms of synchronization, which in turn might lead to wrong results for subsequent analyses, e.g., data race detection. The strategy for communication between the tool threads is non-blocking, i.e., whenever one

tool thread sends analysis data to another, it is not required to block until the data is received at the destination. In general, the implementation has been designed to block as minimal as possible while still delivering correct vector clock results.

### 3.4.3 Piggybacking Vector Clocks

To realize the vector clock exchange mechanism described in the previous sections, some kind of piggybacking mechanism has to be implemented. Essentially, the vector clocks relevant to other processes must be sent along with the corresponding synchronization operation. For point-to-point messaging, especially in MPI, there are various ways of how this piggybacking could be implemented as discussed in [86], namely by (1) appending the additional data to the payload of the message, (2) using MPI data types to encode the additional data, or (3) sending separate messages with the piggyback data. The results of Schulz et al. [86] show that the overhead introduced by the different techniques heavily depends on the hardware, the MPI implementation, and the usage scenario, so there is no single best choice.

For the vector clock exchange mechanism, the option of appending the piggyback data to the message's payload would be well-defined only for point-to-point messaging, not for collective operations or resource-bound synchronization. The option using MPI data types is limited to MPI programs and would not work with SHMEM and GASPI. Therefore, the only viable option is sending separate messages with the piggyback data, i.e., vector clocks, because it allows for the highest flexibility in terms of the implementation. In particular, the separation of application communication and tool communication aligns with GTI's infrastructure, which uses separate communication channels for tool data.

### 3.4.4 Tracking Handles With MUST

In all three programming models, handles represent specific contexts or groups of processes. For example, MPI uses communicators to identify groups of processes. Communicators are represented as opaque objects at runtime. SHMEM defines a similar concept called teams. All those handles must be translated, e.g., communicators to the corresponding process groups at runtime to perform the required vector clock exchange. Another example is an MPI request handle that can be used to complete non-blocking operations. Request handles must be tracked to understand the completion semantics of calls such as *MPI\_Wait*.

MUST [36] already comes with a rich tracking infrastructure integrated for MPI handles (communicators, requests, data types, windows), that is utilized in the MPI vector clock wrapper in the translation to the vector clock primitives. In the context of this thesis, I extended the infrastructure to support SHMEM teams and segments, as well as GASPI segments.

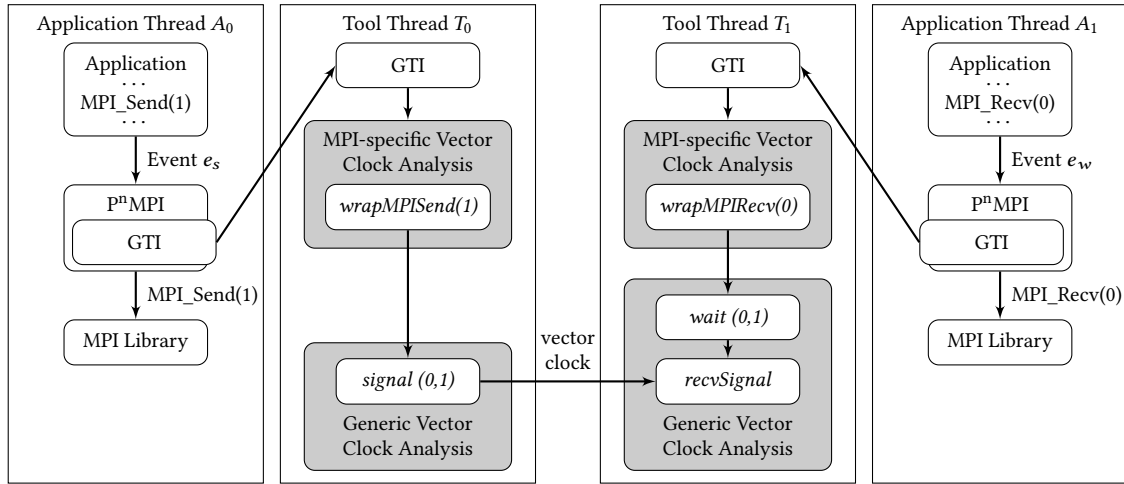


Figure 3.11: Analysis workflow for the vector clock exchange of two processes synchronizing via *MPI\_Send* and *MPI\_Recv*. The gray boxes depict the modules implemented into the existing GTI infrastructure. Adapted from [89] and [103].

### 3.4.5 Analysis Workflow

The vector clock exchange mechanism consists of (1) an analysis module that implements the generic primitives of the clock exchange and (2) model-specific wrapper modules that translate the concrete calls of a programming model (MPI, SHMEM, GASPI) to the generic primitives. Both the generic module and the model-specific modules run on each tool thread  $T_i$  associated with each process  $P_i$ , so each tool thread  $T_i$  maintains the vector clock  $V_i$  of process  $P_i$ .

Figure 3.11 shows the analysis workflow at the example of two processes synchronizing via *MPI\_Send* and *MPI\_Recv*. The application thread  $A_0$  first calls *MPI\_Send(1)* that is wrapped via  $P^N$  MPI and GTI to an event  $e_s$ , sent to the tool thread  $T_0$  via GTI. The tool thread  $T_0$  then reacts to the *MPI\_Send(1)* by wrapping it in the model-specific wrapper to the primitive *signal(0,1)* that is further processed by the generic vector clock module. The *signal(0,1)* primitive at tool thread  $T_0$  first increases the local clock entry  $V_0[0]$  by one and then sends out the current vector clock  $V_0$  to tool thread  $T_1$  via GTI. Then, tool thread  $T_0$  finishes the processing of the event and acknowledges that to the application thread  $A_0$ , which in turn finally passes the *MPI\_Send(1)* event to the MPI library for processing and continues execution.

When tool thread  $T_1$  receives the vector clock  $V_0$  with the signal event from  $T_0$ , it is processed in a *recvSignal* function. If the *MPI\_Recv(0)* at  $A_1$  has already been called and mapped to a *wait(0,1)* event that matches the *signal(0,1)* from  $T_0$ , then the received vector clock  $V_0$  is merged with the current vector clock  $V_1$  at  $T_1$ . If the matching *MPI\_Recv(0)* has not been called so far, then the received clock is buffered at  $T_1$  for later processing.

The wrapping of model-specific calls also requires translating model-specific concepts, such as MPI communicators, requests and SHMEM teams. For that, the MUST resource infrastructure is utilized as discussed in Section 3.4.4.

The following three subsections summarize the implementation considerations for the different kinds of synchronization in the clock exchange model. More details on the implementation are described in [103].

### Point-to-Point Synchronization

For point-to-point synchronization, the workflow depicted in Figure 3.11 has already been discussed as a representative for such an exchange. Message-based exchanges often support sending messages via independent queues or channels. For example, MPI messages may be distinguished using tags. This information also has to be encoded in the vector clock exchange to ensure that the matching of vector clocks is in line with the application’s message matching. To distinguish different such different identifiers, a *queue* identifier is sent along with *signal* and *wait* events.

### Collective Synchronization

As discussed in Section 3.3.2 in detail, *all-to-one* synchronization maps to an elementwise maximum reduction of the vector clock, *one-to-all* synchronization to a broadcast of the vector clock, and *all-to-all* synchronization to a hybrid of an all-gather and all-reduce operation, depending on whether a process participates in the collective exchange or not. For an efficient clock exchange between the tool threads, the vector clock tracking implements broadcast, reduction, and gather operations using a binomial tree on top of the GTI communication channels, specifically tailored to the vector clock exchange.

### Resource-Bound Synchronization

The resource-bound synchronization requires a designated deposition process  $D(R)$  to be specified for each resource  $R$ . In the implementation, the process that created or “owns” the resource is the deposition process, e.g., an MPI RMA window on a process  $P_i$  (which can be locked by another process) has  $P_i$  as the deposition process. If a resource does not have an owning process which is, for example, the case for SHMEM locks, a default process such as  $P_0$  is used. To avoid contention on a single process, such resources might be distributed among different processes in a round-robin fashion.

When process  $P_i$  signals to a resource  $R$ , it sends its current vector clock  $V_i$  via GTI to the deposition  $D(R)$  where it is merged with the stored vector clock. When process  $P_i$  waits for a resource  $R$ , it uses a request-response protocol to fetch the vector clock from  $D(R)$ : It sends a request for the vector clock stored for  $R$  to deposition  $D(R)$  which in turn

responds with the vector clock of  $R$  to  $P_i$ . To reduce latency, one-sided communication might be used to fetch and deposit the vector clocks, but an extension of GTI would be required to support such communication.

#### 3.4.6 User Annotations

The implementation might miss synchronization if (1) new features or routines are added to a programming model that are not yet mapped or (2) if synchronization cannot be associated with a call in the programming model. For the latter one, an example is the polling on a memory location in MPI RMA using plain local memory accesses, as shown in Figure 3.12. The local memory accesses to location  $X$  in the while-loop would lead to synchronization when the while-loop is left. It would be infeasible in terms of performance to treat any local memory load access as a potential synchronization point. Therefore, this kind of synchronization is ignored in the implementation.

To still support scenarios with new synchronization routines or unsupported synchronization, the implementation provides a user annotation API inspired by the ThreadSanitizer annotation API [94]. The user can insert additional annotation calls in their source code to ensure manual synchronization mapping. In particular, the following calls are provided as defined in [90]:

- *AnnotateTick()*: Increment the local vector clock entry by one.
- *AnnotateProcessSignal(target, queue)*: Send signal to given target process using the given queue identifier.
- *AnnotateProcessWait(origin, queue)*: Wait for signal from given origin process on the given queue and merge the own vector clock with the received vector clock.
- *AnnotateResourceSignal(resource, deposition)*: Deposit current vector clock to the given resource at the given deposition process.
- *AnnotateResourceWait(resource, deposition)*: Fetch vector clock from the given resource at the given deposition process and merge it with the own vector clock.

For process-bound synchronization, the *target* and *origin* represent the process IDs the vector clock should be sent to or received from. The *queue* identifier may be used to separate synchronization, e.g., for MPI communicators or SHMEM contexts. For resource-bound synchronization, the *deposition* is a designated process ID where the vector clock of the given *resource* should be stored and fetched from.

In Figure 3.12, the synchronization through polling on memory location  $X$  is annotated using *AnnotateResourceSignal* and *AnnotateResourceWait* such that the vector clocks are exchanged correspondingly.

| Process P0   | Process P1  |
|--|---|
| <pre> MPI_Win_lock(P1, EXCLUSIVE) <b>AnnotateResourceSignal(X, P1)</b> MPI_Put(1, P1, X) MPI_Win_unlock(P1) </pre> | <pre> window location X (initialized with 0)  while (X == 0) {     // MPI progress (not depicted) } <b>AnnotateResourceWait(X, P1)</b> </pre> |

Figure 3.12: Polling example in MPI RMA with user annotation functions: Process  $P_0$  writes the value 1 to memory location  $X$  at process  $P_1$ . This releases process  $P_1$  that is polling in a while-loop for  $X$  to become 1, so  $P_1$  effectively waits for an event of  $P_0$ . This synchronization is annotated with *AnnotateResourceSignal* and *AnnotateResourceWait*. Adapted from [89].

### 3.4.7 Summary

The presented on-the-fly vector clock exchange can track process synchronization in distributed-memory programs at runtime by utilizing the MUST correctness checking framework with the communication layer GTI. It implements all synchronization primitives defined in Section 3.3 and supports a large variety of process-bound and resource-bound synchronization methods. Wrapper modules translate the concrete synchronization routines in MPI, SHMEM, and GASPI to abstract synchronization primitives. The implementation can record nearly all the established process synchronization in those three models. An exception to that is synchronization that cannot be attributed to a routine in a programming model, such as polling on memory location in MPI RMA. In that case, the user can manually enrich the source code with the intended synchronization behavior through an annotation interface. For future extensions, adding support for another programming model only requires adding an additional wrapper module that translates the concrete routines to the abstract primitives of the exchange model.

## 3.5 Overhead Evaluation

This section presents an overhead evaluation of the implemented vector clock exchange on the SPEC MPI 2007 benchmark suite [99] to get an impression of the overhead. The SPEC suite consists of multiple compute-intensive benchmarks parallelized with MPI. The original evaluation [103] has been performed on the RWTH cluster CLAIX-2018 [17]; the following subsections describe a reevaluation of the results on the recent RWTH cluster CLAIX-2023 [18]. The results on CLAIX-2023 are qualitatively similar to those of CLAIX-2018, showing that the vector clock exchange works on different clusters and different software stacks.

### 3.5.1 Experiment Setup

The experiments were performed on CLAX-2023 [18] consisting of a total of 630 nodes connected via InfiniBand. Each node is equipped with two Intel Sapphire Rapids 8468, in total 96 cores, with SMT disabled and in total 256 GB main memory available. The benchmarks were executed with Intel MPI 2021.6 and used the Intel C/C++ and Fortran compiler in version 2021.6.0. For reproducibility, the evaluation is implemented using the JUBE Benchmarking Environment<sup>2</sup> and is available with all configurations and data in the thesis artifact provided in Appendix A.

From the SPEC benchmark suite in version 2.0, all applications that have a *large* input set (*lref*) were chosen as the test set to run large-scale measurements with 192, 384, and 768 application processes. The only excluded benchmark is *121.pop2* due to timeouts when running with and without tool. The benchmarks use a mixture of MPI point-to-point and collective communication, i.e., process-bound synchronization. For the baseline measurement, the total execution time  $T_{Baseline}$  of the application is measured without any attached tool. For the tool measurement, the total execution time  $T_{Tool}$  with the vector clock exchange performed is measured. Both time measurements are performed three times and the median out of the three runs is chosen. Finally, the slowdown factor  $S = \frac{T_{Tool}}{T_{Baseline}}$  is computed.

The vector clock exchange requires an additional tool thread for each MPI application process. Therefore, the total number of 96 physical cores per node is split into two halves: The first half of 48 cores is used for the application itself, the second half of 48 cores is used for the tool threads. A spread binding is chosen to pin the application's cores to even core numbers and the corresponding tool threads to odd core numbers. The baseline run only utilizes 48 of the total 96 physical cores per node, while the tool run uses the spare 48 cores for the tool threads. Considering that the application uses 48 physical cores per node, the number of nodes for 192, 384, and 768 application processes is 4, 8, and 16, respectively.

### 3.5.2 Results and Discussion

Figure 3.13 shows the slowdown for the different SPEC benchmark applications compared to the baseline run. For most applications, the slowdown ranges from 1.2x to 3.2x. Exceptions are *143.dleslie* where the vector clock exchange slows down the execution up to 17x and *128.GAPgeofem* with a slowdown of up to 19x for 768 application processes. In general, increasing the number of application processes increases the slowdown for all benchmarks.

The main factors influencing the slowdown have been investigated in detail in Felix Tomski's master thesis [103]. Generally, the higher the number of MPI calls issued by the application in a specific time frame, the higher the tool overhead. For example,

---

<sup>2</sup><https://www.fz-juelich.de/en/ias/jsc/services/user-support/software-tools/jube>

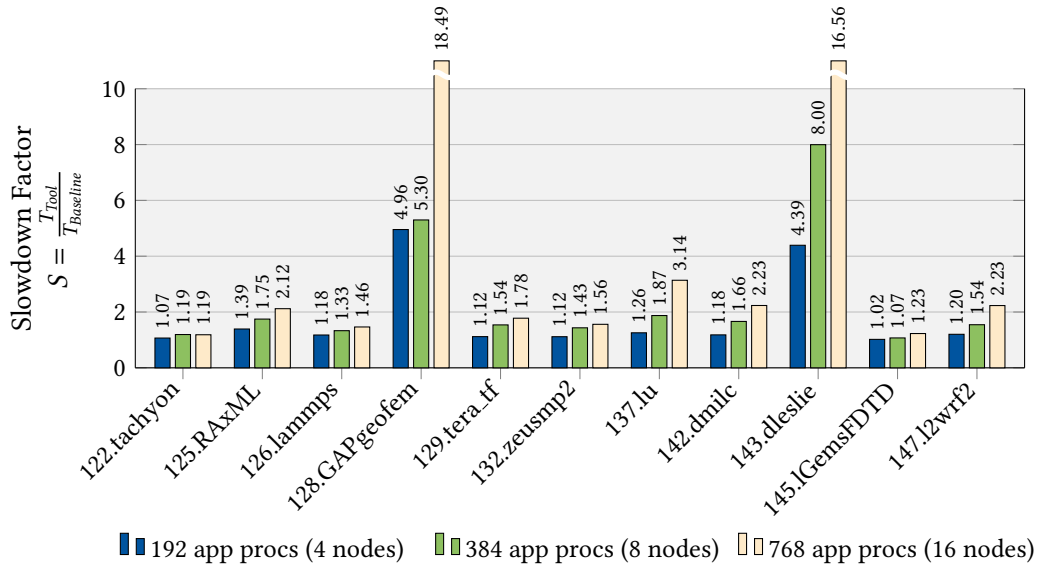


Figure 3.13: Slowdown factors of the vector clock exchange compared to the baseline run for the SPEC MPI benchmarks using tool threads on CLAIx-2023.

128.GAPgeofem issues thousands of MPI collective calls per process per second, which is far higher than the benchmarks with lower overhead. Similar results have been reported by Hilbrich et al. [37]. Since each collective call triggers a collective vector clock exchange in the tool, this causes a high overhead. Moreover, the number of collective MPI calls per process in 128.GAPgeofem increases linearly with the number of processes, explaining the significant increases of the slowdown with a higher number of processes.

The flexibility of the GTI tool infrastructure allows running the vector clock exchange also in separate tool processes: Instead of running a tool *thread* within each application process, an alternative configuration is running a dedicated tool *process* spawned along with each application process. The main difference between those configurations is the internal GTI communication protocol: While the communication with the tool thread uses shared memory, the communication with the tool process uses MPI. Using tool processes has beneficial impacts on the performance, as shown in Figure 3.14. The overhead of all benchmarks, in particular 128.GAPgeofem and 143.dleslie, is reduced. The main reason is that the GTI communication protocol for tool threads relies on a manually written shared-memory data exchange, while the protocol for tool processes relies on the efficient message exchanges of the MPI library. If possible, tool processes should, therefore, be favored. However, depending on the correctness analysis implemented on top of the vector clock exchange, using tool threads might still be necessary despite their inferior performance: The RMA race detector implementation discussed in Chapter 5 requires tool threads due to its architectural design relying on the interaction with ThreadSanitizer.

Summing up the SPEC MPI benchmarks results, it shows that the on-the-fly vector clock exchange is applicable to large-scale real-world applications with low overhead in most cases. Each invoked point-to-point MPI routine requires piggybacking an additional

### 3 Clock-Based Synchronization Analysis for Distributed-Memory Programs

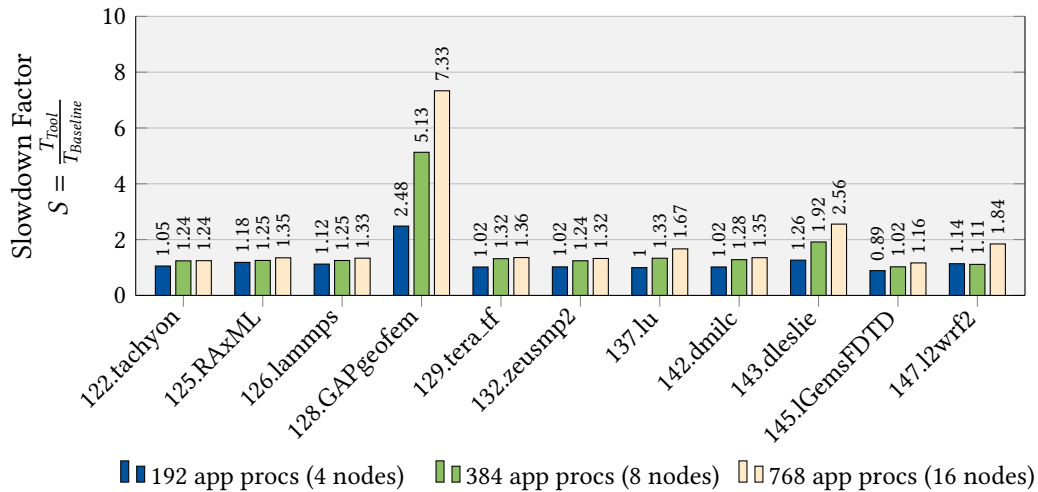


Figure 3.14: Slowdown factors of the vector clock exchange compared to the baseline run for the SPEC MPI benchmarks using tool processes on CLAIX-2023.

message with the vector clock that must be merged with the vector clock at the destination process and each invoked collective requires a collective exchange of the vector clocks between all processes. This information must travel through the tool infrastructure, contributing to the overhead. Further, to always have up-to-date vector clocks, processes that have to wait for a vector clock from another process must stall the application’s execution. Otherwise, dependent analyses might work with outdated vector clocks which is especially harmful for the purpose of using the vector clocks for correctness analyses such as data race detection. Still, whenever possible, the exchanged vector clocks are processed asynchronously, e.g., a tool thread may proactively process and queue a received vector clock before the application performs the corresponding synchronization. This is why outsourcing the vector clock management to separate tool threads or processes is still reasonable for reducing the overhead.

## 3.6 Hybrid Parallelism

The clock exchange model described in this chapter assumes a single-threaded execution with  $N$  processes. However, all considered programming models, MPI [28, §11], SHMEM [19, §9.2], and GASPI [27, §3], support multithreading, i.e., when thread support is requested at startup, threads within the same process may call all routines at any time (concurrently). An extension to support such multithreaded executions in the vector clock exchange raises additional challenges: First, multithreading in shared-memory programs typically creates and destroys threads dynamically at runtime through fork-join constructs. Thus, hybrid vector clocks need to cope with a dynamic number of vector clock entries, either by statically allocating a sufficient number of clock entries at the beginning or by providing a functionality to grow and shrink dynamically. Second, threads

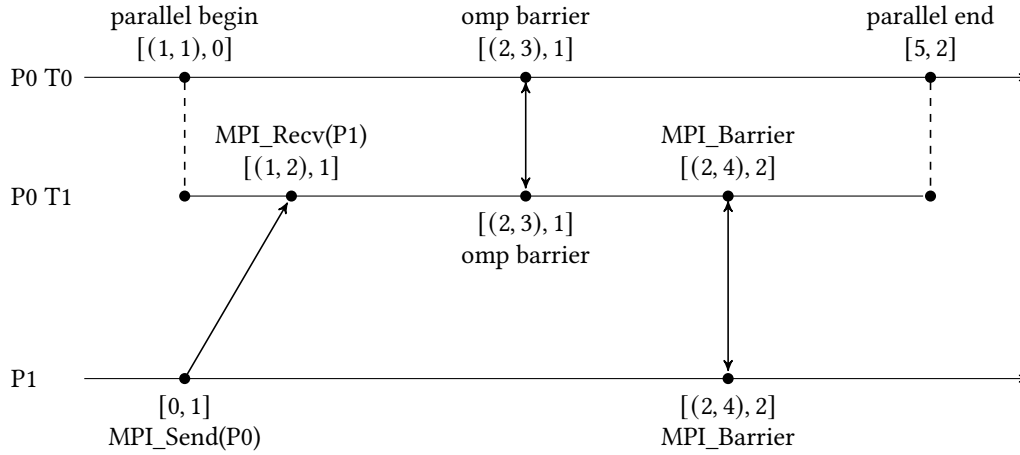


Figure 3.15: MPI+OpenMP synchronization example with nested (hybrid) vector clocks:  $P_0$  creates a parallel region with two threads in which  $T_1$  synchronizes with  $P_1$  via  $MPI\_Recv$ . This synchronization is then propagated to  $T_0$  by an OpenMP barrier between the two threads at  $P_0$ . Finally, an  $MPI\_Barrier$  synchronizes  $T_1$  and  $P_1$  in both directions. Adapted from [75].

of one process might synchronize *individually* to other processes and then transitively propagate that causality to the other threads through internal (thread) synchronization, as shown in Figure 3.15. Therefore, shared-memory synchronization between threads also has to be fully captured to get a complete picture of the happened-before relation in the case of hybrid parallelism.

The previously named challenges have been addressed in Cornelius Pätzold’s master thesis [75]. It investigates two variants of coupling the distributed-memory parallelism with the fork-join parallelism of OpenMP in a hybrid vector clock. The first variant uses *nested vector clocks* as a recursive data structure where each entry of the vector clock may be (1) a single integer or (2), in turn, consist of another vector clock. Any start of a parallel region with  $M$  threads replaces the previous integer entry with a vector of  $M$  entries. An example of an OpenMP parallel region in combination with MPI synchronization is shown in Figure 3.15. The vector clock comparison operation is correspondingly extended to consider such nested structures. Alternatively, in a second variant, hybrid vector clocks may be represented as *partially ordered logical clocks* [24], a set of key-value pairs  $\{(k_1, t_1), \dots, (k_m, t_m)\}$  where each key  $k_i$  is a thread identifier that is unique among all (globally) active threads and  $t_i$  is a single integer. The set of key-value pairs can grow dynamically with the number of threads. The merging and comparison operations are adapted to work on the key-value structure.

In the context of Cornelius Pätzold’s thesis [75], the original vector clock implementation has been extended to a hybrid clock variant supporting the combination of MPI, SHMEM, and GASPI with OpenMP. The implementation manages a local OpenMP vector clock recording the shared-memory synchronization coupled with the process vector clock recording the distributed-memory synchronization, resulting in the hybrid clock. To

capture the OpenMP synchronization, an analysis present in the data race detection tool Archer [3] based on the OpenMP Tools Interface [7, §19] is reused.

The two variants of hybrid vector clocks, the nested vector clock and the clock with key-value pairs were implemented and evaluated. Both implementations showed significantly higher slowdown factors than the original single-threaded vector clock exchange. The main reason is a design limitation of the vector clock exchange: *All* synchronization events within a process are redirected to a *single* tool thread that manages *all* vector clocks for all threads, thus leading to a performance bottleneck on a high number of application threads. Further, when different threads perform independent collective operations or message exchanges concurrently, piggybacking with separate messages may deliver non-deterministic results or deadlocks in the hybrid vector clock exchange. Future extensions may address the performance bottleneck and deadlocks by offloading some analysis parts to the application threads or using multiple tool threads.

## 3.7 Use Cases

Tracking the happened-before relation, provided by the vector clock exchange model, is a significant cornerstone of the RMA race detection model discussed in Chapter 4. The concrete details on how vector clocks are applied to RMA race detection and finally implemented can be found in Chapter 4 and Chapter 5. In the following, two additional application scenarios of vector clocks, namely MPI I/O race detection and thread-level concurrency checks, show that such a causality tracking mechanism is also useful for other correctness checks.

### 3.7.1 MPI I/O Race Detection

MPI provides a portable I/O abstraction layer [28, §14] for the specific needs of parallel distributed I/O access in large clusters. This includes abstractions to partition data among processes, collective read and write operations on files, strided access, and asynchronous I/O. MPI I/O provides different data access routines based on (1) explicit offsets, (2) individual file pointers, and (3) shared file pointers. Processes might have different *views* on the data whose displacements and strides are defined in terms of MPI datatypes.

Data races on files may occur when two conflicting concurrent file accesses occur without proper synchronization. Two accesses to the same file are conflicting if the accessed file locations overlap and at least one is a write access. With MPI I/O, special care is required when defining the file views (displacements, strides) based on MPI data types to avoid unintended conflicting accesses. Synchronization between conflicting accesses can be ensured by opening files in atomic mode or using consistency and synchronization calls. For the latter variant, coordinated access is achieved by (1) ensuring consistency

```

1 int rank;
2 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
3
4 MPI_File fh; // file handle
5
6 // open file on both ranks
7 MPI_File_open(MPI_COMM_WORLD, "filename",
8               MPI_MODE_RDWR | MPI_MODE_CREATE,
9               MPI_INFO_NULL, &fh);
10
11 // set the same view
12 MPI_File_set_view(fh, 0, MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
13
14 if (rank == 0) {
15     int mydata[5] = {0, 1, 2, 3, 4};
16     MPI_File_write_at(fh, 0, mydata, 5, MPI_INT, MPI_STATUS_IGNORE); // write 5 integers
17
18     MPI_File_sync(fh); // flush to storage
19     MPI_Barrier(MPI_COMM_WORLD); // process synchronization
20     MPI_File_sync(fh); // make visible to all processes
21 } else if (rank == 1) {
22     MPI_File_sync(fh);
23     MPI_Barrier(MPI_COMM_WORLD); // process synchronization
24     MPI_File_sync(fh); // ensure visibility
25
26     int mydata[5];
27     MPI_File_read_at(fh, 0, mydata, 5, MPI_INT, MPI_STATUS_IGNORE); // read 5 integers
28 }

```

Figure 3.16: MPI file I/O example. Adapted from [28, Example 14.5].

through calls of *MPI\_File\_sync* and (2) process synchronization through *MPI\_Barrier* or other means of synchronization.

Figure 3.16 shows how a conflicting write access by  $P_0$  and a read access by  $P_1$  are synchronized using the “sync-barrier-sync” construct. The first (collective) call to *MPI\_File\_sync* ensures that the data written by  $P_0$  is flushed to the underlying storage device. The *MPI\_Barrier* guarantees that the write call of  $P_0$  happens before the read call of  $P_1$ . The second (collective) call to *MPI\_File\_sync* guarantees that  $P_1$  will observe the data written by  $P_0$ . Since the call to *MPI\_File\_sync* is collective, it is required that *both* MPI processes call it before and after the barrier. Omitting the calls to *MPI\_File\_sync* before or after the *MPI\_Barrier* or omitting the *MPI\_Barrier* itself will lead to a data race on the corresponding file location with undefined results [28, §14.6.1].

The complexity of MPI I/O access patterns combined with the involved consistency and synchronization constructs showcase the need of appropriate tool support to aid users in debugging issues with MPI I/O. The master thesis of Paul Lambrich [54] provides an initial view of different erroneous usage scenarios in MPI I/O. It concentrates on measures to detect data races due to concurrent conflicting I/O accesses. The detection model captures the concurrency of MPI I/O calls with vector clocks and combines it with the consistency semantics, e.g., provided by *MPI\_File\_sync*. The implementation has been integrated into MUST and reuses the vector clock exchange mechanism. For each MPI I/O access, the vector clock is recorded and sent to a centralized tool process that tracks

### 3 Clock-Based Synchronization Analysis for Distributed-Memory Programs

```
1 int provided = -1;
2 MPI_Init_thread(&argc, &argv,
3     MPI_THREAD_SERIALIZED, &provided);
4 // thread-level check omitted here
5
6 #pragma omp parallel num_threads(2)
7 {
8     #pragma omp master
9     { MPI_Send(...); }
10    #pragma omp single
11    { MPI_Recv(...); }
12 }
13
14 MPI_Finalize();
```

```
1 int provided = -1;
2 MPI_Init_thread(&argc, &argv,
3     MPI_THREAD_SERIALIZED, &provided);
4 // thread-level check omitted here
5
6 #pragma omp parallel num_threads(2)
7 {
8     #pragma omp master
9     { MPI_Send(...); }
10    #pragma omp barrier
11    #pragma omp single
12    { MPI_Recv(...); }
13 }
14 MPI_Finalize();
```

(a) Violation of `MPI_THREAD_SERIALIZED` level.

(b) Serialized execution with OpenMP barrier.

Figure 3.17: MPI+OpenMP example with thread-level violation (left) and the fixed source code (right).

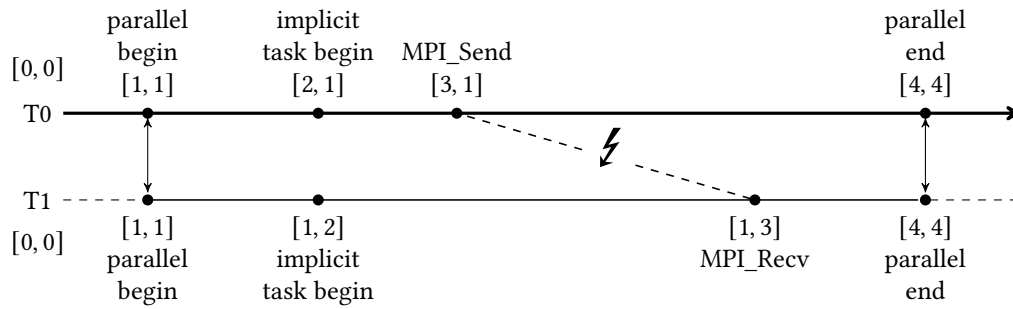
all I/O access intervals in an Interval Skip List (ISL) [34]. The ISL can find overlapping intervals in  $O(\log N)$  time with  $N$  as the number of intervals stored in the list.

The prototype implementation worked on basic test cases but still has to be evaluated on large real-world programs. A significant concern is the centralized detection, which will become a bottleneck if all processes send their accessed intervals to a single tool process. The main reason for the centralized design is that file storage locations are not owned by any process, compared to allocated memory regions in shared memory or RMA. Thus, for a distributed analysis, processes would have to agree on which process is responsible for managing the accesses to which file (or file part). This, however, significantly complicates the exchange mechanism of accessed intervals that would have to be split up depending on which process is responsible for a certain file part.

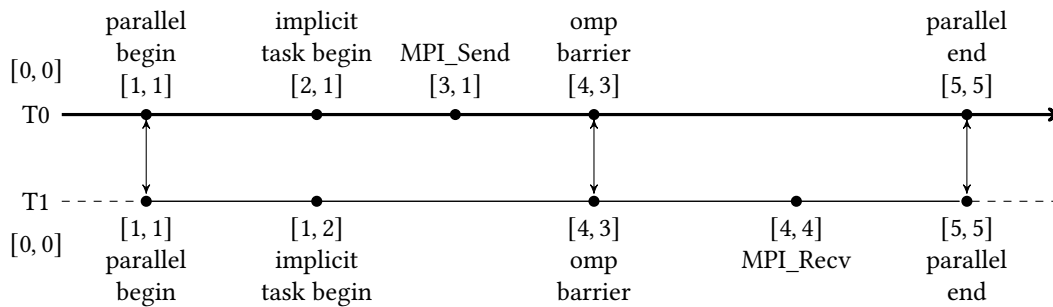
The results of Paul Lambrich’s thesis showcase the flexibility of the vector clock mechanism. Initially intended for the use case of RMA race detection, it can also be applied to correctness analysis of issues in other programming paradigms of distributed-memory programs.

#### 3.7.2 Thread-Level Concurrency Checks for MPI+OpenMP

As a byproduct of the hybrid vector clock representation in Section 3.6, the captured OpenMP clock allows to check if MPI routines performed by different threads (in the same process) are called concurrently. Cornelius Pätzold discussed in his master thesis [75] and a subsequent publication [76] different classes of errors and code smells in MPI+OpenMP programs due to wrong thread interaction. The classes of errors include `MPI_THREAD_SERIALIZED` violations and concurrent collective calls on the same communicator, while the classes of code smells include concurrent receive and probe calls. Both classes and how to detect those situations will be discussed in the following.



(a) The calls to *MPI\_Send* of  $T_0$  and *MPI\_Recv* of  $T_1$  are concurrent as indicated by the vector clocks.



(b) The calls to *MPI\_Send* of  $T_0$  and *MPI\_Recv* of  $T_1$  are serialized: *MPI\_Send* happens before *MPI\_Recv*.

Figure 3.18: Vector clock representation of the code examples in Figure 3.17. Adapted from [76].

When initializing an MPI program, the developer must explicitly request thread support from the MPI library with a corresponding thread level. The differentiation between the thread-support levels *MPI\_THREAD\_SERIALIZED* and *MPI\_THREAD\_MULTIPLE* is in particular of interest. While the latter allows the MPI processes to use multithreading and any thread may perform MPI calls at any time, the former requires the MPI calls invoked by different threads to be serialized, i.e., only one thread at a time is allowed to make MPI calls. For the *MPI\_THREAD\_SERIALIZED* case, the developer is responsible for ensuring the serialization through synchronization between the threads, e.g., via OpenMP critical sections. Violating the rules of the provided thread-support level leads to undefined behavior. Figure 3.17a shows an example of a parallel region with two threads where the master thread executes an *MPI\_Send* due to the *master* construct, and the subsequent code in the *single* construct may be executed by the other thread. Since the *master* construct of OpenMP does not imply a barrier, both MPI calls may be executed concurrently, although the requested thread-level is *MPI\_THREAD\_SERIALIZED*. Figure 3.17b depicts the fixed version with an OpenMP barrier between the constructs.

Assuming that the thread level has been set correctly to *MPI\_THREAD\_MULTIPLE*, some MPI calls still should not be invoked concurrently: For example, using the same communicator in MPI collectives concurrently from different threads is forbidden. Further, non-erroneous but probably unintended situations include concurrent receive calls that

specify the same envelope, leading to a non-deterministic matching of the messages to the corresponding threads. Similarly, concurrent probing with *MPI\_Probe* (instead of *MPI\_Mprobe*) and receiving might lead to deadlocks, especially if one thread probes for a message that another thread may receive in between.

All discussed concurrency situations can be detected by analyzing the causality of MPI calls using the OpenMP vector clock. Figure 3.18 shows how executing the codes in Figure 3.17 would be translated to OpenMP vector clock events. The OMPT events [7] increase the vector clock and trigger corresponding vector clock exchanges. In Figure 3.18a, the *MPI\_Send* call invoked by  $T_0$  is concurrent to the *MPI\_Recv* call invoked by thread  $T_1$  since the vector clocks are incomparable. In Figure 3.18b, the two MPI calls are serialized using an OpenMP barrier, correspondingly reflected in the vector clocks.

The clock-based concurrency analysis for MPI+OpenMP programs has been implemented as a prototype in the context of Cornelius Pätzold’s master thesis [75]. It uses the FastTrack algorithm [26] to analyze the concurrency of different MPI calls efficiently. In particular, it avoids storing the complete full vector clock and only requires  $O(1)$  integer comparisons instead of  $O(m)$  for  $m$  threads on any relevant MPI event. Details of the approach are discussed in [75] and [76].

## 3.8 Related Work

Tracking of causality in parallel program executions is an elemental requirement for many correctness checks. Therefore, the vector clocks, as defined by Fidge [24] and Schwarz and Mattern [87], are used in various areas of correctness checking that will be discussed and related to the vector clock model presented in this thesis in the following.

### 3.8.1 Shared-Memory Race Detection

Many data race detectors for shared-memory programs [3, 26, 77, 94, 95] rely on happened-before analysis that is tracked on-the-fly with vector clocks, similar to the approach presented here. In shared-memory programming, synchronization is predominantly resource-bound (mutexes, locks), so the vector clocks are typically associated with resources. In distributed-memory programs, there are various ways of process-bound synchronization through message exchanges and collectives that have to be tracked in addition to resource-bound synchronization. The vector clock model, as defined and implemented in this thesis, supports both kinds of synchronization patterns. From the implementation perspective, the exchange and management of vector clocks in a distributed-memory system have significantly different requirements than in a shared-memory system. In shared-memory systems, all threads have a shared view on all vector clocks and resources and can directly access it, whereas distributed-memory system have separate private memory regions where the local vector clocks are stored. Especially

for process-bound synchronization, the vector clock has to be piggybacked instead of written into a shared memory location. Therefore, other methods for efficient vector clock exchanges have to be used and it is not possible to reuse existing implementations for shared-memory systems.

### 3.8.2 Alternative MPI Matchings

Besides race detection, logical clocks might be used to explore alternative execution paths. DAMPI [108] is a dynamic formal verifier for MPI programs that tests through all possible execution paths by considering all sources of non-determinism (e.g., wildcard-receives or other non-deterministic receives and probes). Thereby, it detects deadlocks and resource leaks on the explored execution paths. To find all possible message matchings, DAMPI first executes the MPI program once to record an initial execution of the program. While executing the MPI program, it piggybacks with every message a logical clock in a separate message, similar to the approach presented in this thesis. DAMPI supports both piggybacking a complete vector clock or a Lamport clock. For DAMPI, the Lamport clock has been proven sufficient in most cases and is more efficient in terms of transmission size and comparison. By comparing the logical clocks of messages with the clock of wildcard receive calls, DAMPI can find alternative message sends that happened before and have a matching envelope. Those serve as candidates for an alternative execution path. After the initial run, DAMPI replays the execution and enforces the alternative matchings to explore another execution path.

Compared to the approach presented in this thesis, DAMPI achieves a similar goal of tracking logical clocks at runtime. However, the DAMPI clock exchange is limited to process-bound synchronization (point-to-point, collectives), any resource-bound synchronization (locks, polling) is not supported. Further, the model and the implementation are focused on MPI programs, while the approach presented here provides a generalized model that has also been proven to work with SHMEM and GASPI programs.

### 3.8.3 Post-Mortem RMA Race Detection

MC-CChecker [22] is a race detector for MPI RMA that uses vector clocks for post-mortem race detection. It first executes the program to record all relevant events in a trace. A post-mortem analysis uses a timestamping system to label all events with vector clocks and finally detect conflicting concurrent remote memory accesses. The vector clocks allow MC-CChecker to understand transitive synchronization effects that have been ignored by the predecessor MC-Checker [16], which relied on a DAG to detect conflicting and concurrent accesses. Instead of using the usual vector clock with  $N$  entries for  $N$  processes, MC-CChecker uses encoded vector clocks [52] where a single large number represents a vector clock and each process  $P_i$  is associated a unique prime number. A vector clock tick on a process is then a multiplication with a unique prime

number and vector clocks can be compared by performing a division operation, as detailed in [52]. Since the vector clock represented by a single large number can grow extensively, measures to reset the encoded vector clock at defined points have been discussed in further research [78].

MC-CChecker models the synchronization similarly to the model presented in this thesis but only considers process-bound synchronization. Any resource-bound synchronization (locks, polling) is not modeled and, therefore, is not supported. Further, the approach is limited to MPI RMA programs and only performs the timestamping in a post-mortem analysis, not at runtime as the approach presented in this thesis. The main drawback of a post-mortem analysis is that the synchronization analysis can only be performed at the end of the program execution, so any correctness issue in the program can only be detected after termination of the application, which may be infeasible for long-running applications. Further, post-mortem analysis relies on trace files that can grow very large for long-running applications, leading to larger storage requirements and potentially long analysis time.

#### 3.8.4 Other Related Work

There are various other application areas where vector clocks are used. SReplay [80] provides a record-and-replay mechanism for programs with one-sided communication, focused on UPC programs [31]. It records the order of memory accesses for a designated subgroup of processes by using vector clocks: Every process  $P_i$  in the subgroup maintains a local vector clock  $V_i$ , and for each shared address  $x$  (in the global address space), an access vector clock  $V_x^a$  and a write vector clock  $V_x^w$  is maintained. On every memory access by process  $P_i$ , it ticks its local vector clock  $V_i$  and updates the corresponding vector clock  $V_x^a$  or  $V_x^w$  associated with the memory address. In the (partial) replay phase, SReplay reproduces for the recorded sub-group of processes the recorded order of memory accesses. Thereby, it explicitly delays the execution (if required). The authors outlined different application scenarios of the replay mechanism, such as detecting non-deterministic executions and sequential consistency violations. Thus, SReplay uses vector clocks to record a particular order of memory accesses for replay analysis, but not a synchronization order as proposed in this thesis.

In another recent work [111], the vector clock concept has been extended to perform may-happen-in-parallel analysis for static data race detection on shared-memory programs using pthreads. The static vector clock is represented by sets of instructions (that may be executed by a thread) instead of natural numbers. The partial order representing the may-happen-in-parallel relation is adapted accordingly and differs significantly from the classical vector clock model assumed in this thesis.

## 3.9 Results and Discussion

This chapter introduces a clock-based model to track the synchronization in distributed-memory programs, focused on the programming models MPI, OpenSHMEM, and GASPI. For that, a classification of synchronization in those programming models is provided, distinguishing between process-bound synchronization, i.e., synchronization established by directly addressing processes, and resource-bound synchronization, i.e., synchronization established indirectly with signaling to and waiting for a resource.

The first main contribution is the definition of generalized synchronization primitives that can capture all the different synchronization mechanisms in distributed-memory programs. The set of synchronization primitives consists of *signal*, *wait*, *all-to-all*, *all-to-one*, and *one-to-all*. As the application to the three RMA models shows, those primitives are enough to capture all relevant synchronization behavior. For all procedures defined in the three RMA models, their synchronization semantics is specified in terms of the five synchronization primitives. This, in particular, also includes all the involved synchronization semantics in MPI via point-to-point communication in all different flavors, e.g., synchronous, non-blocking, and persistent communication.

Based on the generalized synchronization primitives, the second main contribution is the definition and implementation of a vector clock exchange suitable for an on-the-fly tracking of synchronization in distributed-memory programs. It enables subsequent correctness analyses to reason at runtime about the synchronization state of the processes, which is a typical requirement for on-the-fly data race detection. The vector clock exchange is implemented as part of the MUST correctness checking infrastructure and supports MPI, OpenSHMEM, and GASPI. To make the analysis scale to larger program runs, the clock exchange is designed to be as efficient as possible, in particular for collective synchronization. The architecture is designed to be extensible to other programming models and a user annotation API is also provided to make the annotation of custom synchronization concepts possible. The application to the SPEC MPI 2007 benchmarks shows that the vector clock exchange is applicable to real-world applications with, in most cases, slowdowns of 1.1x to 3.5x for up to 768 processes.

The defined and implemented vector clock model opens up a large field of (generalized) on-the-fly correctness analyses that were previously not possible. For example, the vector clock exchange is used to perform RMA race detection in MPI RMA, OpenSHMEM, and GASPI programs, which will be discussed in detail in the following chapters. Further, the model has been used to perform data race detection in MPI file I/O and has also been extended to hybrid programs using MPI+OpenMP, showcasing its versatility.



## 4 Event-Based RMA Race Detection Model

In remote memory access (RMA) programs, data races may occur due to improper synchronization of concurrent conflicting accesses. As data races are undefined behavior in most RMA models, the results of such an incorrect program may vary from run to run or depend on the underlying RMA implementation and hardware platform. Therefore, data races are often hidden bugs that only manifest non-deterministically as failures in certain circumstances. The manual detection of such data races is difficult, mainly due to the involved consistency and synchronization semantics of RMA models. Therefore, tool support for race detection in RMA programs would be helpful. While data race detection in shared-memory programs is a well-studied problem with appropriate mature dynamic tool support [3, 26, 77, 94, 95], less effort with some dynamic prototype tools [2, 16, 49, 73] has been spent so far on the detection of races in RMA programs.

This chapter presents a formal model suitable for on-the-fly data race detection in RMA programs. The model provides a generic set of primitives required for race detection in RMA programs by abstracting away the specific properties of MPI RMA, SHMEM, and GASPI. For each primitive, the corresponding semantics are formally defined through inference rules. Two semantic aspects are relevant for race detection in RMA programs: Process synchronization and the completion state of a remote memory access event. The synchronization of events can be captured with the happened-before relation  $\xrightarrow{hb}$  as shown in Chapter 3. Together with the tracking of the completion state, the happened-before relation  $\xrightarrow{hb}$  can be extended to the consistency relation  $\xrightarrow{co}$  which is introduced in this chapter. Based on the consistency relation, a formalization of data races in RMA is discussed. The formalization of programming model semantics, in general, is required to systematically reason on the correctness of programs, as, for example, also done by Bronevetsky and de Supinski [11] for shared-memory parallelism with OpenMP.

The formal model introduced in this chapter defines the consistency order  $\xrightarrow{co}$  that can be recorded at runtime to detect if there is a race between memory accesses in an RMA program. For that, the concurrent regions of an RMA access are specified. Intuitively, it is the time interval of the execution during which an RMA operation's memory access may occur. The concurrent region combines vector clocks with the RMA completion information to represent the consistency order at runtime.

Figure 4.1 recaptures the two different kinds of races in RMA models, local buffer races and remote races, and depicts the relevant concurrent regions of the RMA accesses. If the

## 4 Event-Based RMA Race Detection Model

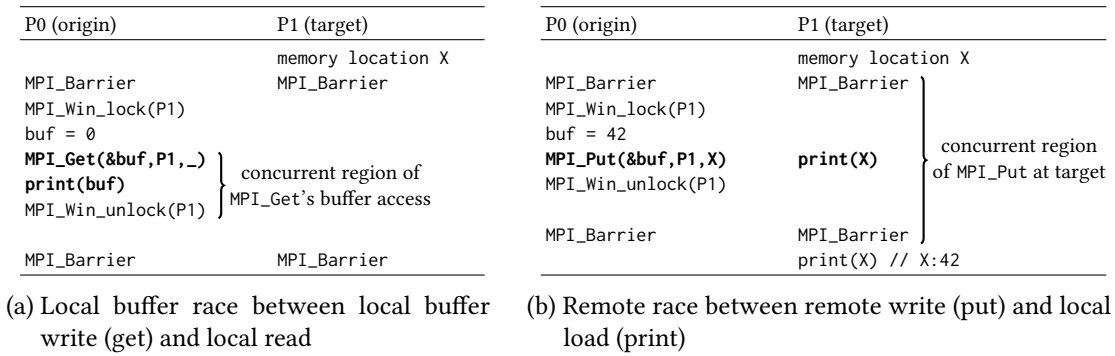


Figure 4.1: Data race examples in MPI RMA using locks. Conflicting statements leading to the race are bold, the concurrent regions define the intervals in which the memory access of the RMA operation may take place. Adapted from [93].

concurrent regions overlap with a conflicting memory access, as in the case of Figure 4.1 with the *print* statement, a data race may occur. The details of this on-the-fly detection algorithm are outlined in this chapter.

The race detection model presented here is based on my previous publication [93], which is a generalized and extended version of the race detection model specifically for MPI RMA that I published in previous works [88, 90].

This chapter is structured as follows: Section 4.1 presents a consistency model capturing the completion semantics of memory accesses in RMA programs, specifically for MPI RMA, SHMEM, and GASPI. It extends the happened-before  $^{hb}$  order to a consistency order  $^{co}$  to reflect the asynchronous nature of RMA operations. The contents of this section are based on my previous publication [93]. Section 4.2 shows how the consistency model can be leveraged to detect the concurrent regions of RMA operations, as shown in my previous publications [89, 93], extended with more examples on corner cases and a formal description of the race detection algorithm. Section 4.3 discusses limitations of the race detection model, only briefly covered in [93], in more detail. Finally, Section 4.4 showcases the generalizability to other RMA models, and Section 4.5 compares the consistency and race detection model with related work.

### 4.1 Consistency Model for RMA Programs

The consistency of memory accesses in an RMA program is an essential property for data race detection. In the following, the system model introduced in Chapter 3 is extended to consider the consistency semantics of RMA programs. This includes the introduction of a consistency order  $^{co}$ , modeled through inference rules, and a formalization of RMA data races.

Table 4.1: Memory events, sets, and helper functions of the model. Adapted from [93].

|  |  |
|--|--|
| $ld(X) \in E$                            | Local memory read from location $X$  |
| $st(X) \in E$                            | Local memory write to location $X$   |
| $rrd(X, P_t) \in E_o$                    | Remote read of location $X$ at target $P_t$ issued at origin $P_o$                       |
| $lwr(X, P_t) \in E_o$                    | Local buffer write to location $X$ resulting from $rrd$ at $P_t$                         |
| $rwr(X, P_t) \in E_o$                    | Remote write to location $X$ at target $P_t$ issued at origin $P_o$                      |
| $lrd(X, P_t) \in E_o$                    | Local buffer read of location $X$ for $rwr$ to $P_t$                                     |
| $L \subseteq E$                          | Set of plain local memory reads ( $ld$ ) and writes ( $st$ )                             |
| $B \subseteq E$                          | Set of local buffer reads ( $lrd$ ) and writes ( $lwr$ ) from RMA operations             |
| $R \subseteq E$                          | Set of remote reads ( $rrd$ ) and writes ( $rwr$ ) from RMA operations                   |
| $B^{rd} \subseteq B, B^{wr} \subseteq B$ | Read events of $B$ , write events of $B$   |
| $R^{rd} \subseteq R, R^{wr} \subseteq R$ | Read events of $R$ , write events of $R$   |
| $M := L \cup B \cup R$                   | Set of all memory events   |
| $org(e) \in P$                           | Process where $e \in E$ originates / is called (origin)                                  |
| $tgt(e) \in P$                           | Target process associated with RMA operation $e \in B \cup R$ or target notify event $e$ |
| $dst(m) \in P$                           | Process whose memory is accessed by $m \in M$ , i.e., where $m$ takes place              |
| $grp(e) \subseteq P$                     | Processes $G \subseteq P$ addressed by completion event $e$ (if applicable)              |
| $addr(m)$                                | Memory address that is accessed by $m \in M$   |
| $sig(e)$                                 | Signal address used in a target notify or target wait event $e$                          |
| $access(e) \in R$                        | Remote access associated with a target notify event $e$                                  |
| $id(e)$                                  | Identifier of memory access / completion event $e$ for individual local completion       |

### 4.1.1 Event Model

The following sections assume a model with  $N$  single-threaded processes  $P_0, \dots, P_{N-1}$  where for each process  $P_i$  the execution is represented by a set of enumerated events  $E_i = \{e_0^i, e_1^i, \dots\}$ . The set of all processes is  $P := \{P_0, \dots, P_{N-1}\}$  and the set of all events is  $E := E_0 \cup \dots \cup E_{N-1}$ . The model assumed here is an extension to the model used in Chapter 3 for the vector clock exchange.

Table 4.1 lists all relevant memory events, event sets, and helper functions of the consistency model. The memory events consider local memory accesses ( $ld$ ,  $st$ ), remote reads and writes ( $rrd$ ,  $rwr$ ), and local buffer reads and writes ( $lrd$ ,  $lwr$ ). Correspondingly, there are sets for the different kinds of events. As discussed in detail in Section 2.2, an RMA operation comprises a remote access to the target and a local buffer access to the origin. Therefore, an RMA operation such as  $put(buf, X, P_t)$  issued at  $P_o$  is split into (1) an event  $lrd(buf, P_t) \in E_o$  representing the local buffer read and (2) an event  $rwr(X, P_t) \in E_o$  representing the remote write. The differentiation between local memory access ( $ld$ ,  $st$ ) and local buffer accesses ( $lrd$ ,  $lwr$ ) is required since the local buffer accesses of RMA operations are non-blocking and have to be completed through designated RMA completion routines.

The helper functions are used in the formalization to model the semantic properties of the events. In particular, the roles of the processes for a given memory access  $m \in M$  have to be clearly specified. The model distinguishes between  $org(m)$ ,  $tgt(m)$ , and  $dst(m)$ .

Assuming the previous local buffer read access  $b := lrd(buf, P_t) \in E_o$ , it is

$$\text{org}(b) = P_o, \text{tgt}(b) = P_t, \text{dst}(b) = P_o$$

and for the remote write  $r := rwr(X, P_t) \in E_o$ , it is

$$\text{org}(r) = P_o, \text{tgt}(r) = P_t, \text{dst}(r) = P_t.$$

The differentiation between  $\text{tgt}(m)$  and  $\text{dst}(m)$  is required since  $\text{tgt}(m)$  refers to the target process that is associated with the corresponding RMA operation, while  $\text{dst}(m)$  refers to the process where the corresponding memory access  $m$  commits. This is why  $\text{tgt}(b) = \text{tgt}(r) = P_t$ , because both  $b$  and  $r$  are associated with the same RMA operation and thus target process, but the memory access of  $b$  commits at  $P_o$ , i.e.,  $\text{dst}(b) = P_o$ , and the memory access of  $r$  commits at  $P_t$ , i.e.,  $\text{dst}(r) = P_t$ .

The following definition formalizes the notion of conflicting accesses in RMA models:

**Definition 4.1** (Conflicting Memory Events, adapted from [93]). *Two memory events  $m_1, m_2 \in L \cup B$  are locally conflicting if the following conditions hold:*

- (1) *Both access memory on the same process, i.e.,  $\text{dst}(m_1) = \text{dst}(m_2)$ ,*
- (2) *both access the same memory address, i.e.,  $\text{addr}(m_1) = \text{addr}(m_2)$ ,*
- (3) *at least one event is a writing access.*

*Two memory events  $m_1 \in R$  and  $m_2 \in M$  are remotely conflicting if the following conditions hold:*

- (1) *Both access memory on the same process, i.e.,  $\text{dst}(m_1) = \text{dst}(m_2)$ ,*
- (2) *both access the same memory address, i.e.,  $\text{addr}(m_1) = \text{addr}(m_2)$ ,*
- (3) *at least one event is a writing access,*
- (4) *at least one event is not an RMA atomic.*

In the following, two memory accesses  $m_1 \in M$  and  $m_2 \in M$  are called conflicting if they are locally or remotely conflicting.

As the model of the vector clock exchange is re-used, also all synchronization events *signal*, *wait*, *one-to-all*, *all-to-one*, and *all-to-all* are part of the event set. Moreover, the following sections assume that the program order  $\xrightarrow{po}$  and happened-before order  $\xrightarrow{hb}$  will be established between (synchronization) events as discussed in Chapter 3.

### 4.1.2 Consistency Inference Rules

The happened-before order  $\xrightarrow{hb}$  is insufficient to classify races in RMA programs because an RMA memory event only represents the call to an RMA routine, not that the underlying RMA operation is completed when the event occurs. In other words, even if an RMA memory event  $r$  happened before another event  $e$ , there is no guarantee that the memory access of  $r$  is finished when  $e$  occurs. A simple example is the local buffer race example in Figure 4.1a where the call to `MPI_Get`, i.e., the `lwr` and `rrd` event, happens before the call to the conflicting `print` function (due to program order), but the underlying local buffer access may still be concurrent to the access of the `print` function. Thus, an extended order is needed to capture the completion of memory accesses ensured by completion calls such as `MPI_Win_unlock`. As detailed in Section 2.2.3, RMA models distinguish *local completion*, meaning that the memory access to the user-specified local buffer is finished, and *remote completion*, meaning that the memory access at the target is finished. The consistency order  $\xrightarrow{co}$  is defined as follows:

**Definition 4.2** (adapted from [93]). *Let  $a \in M$  and  $b \in E$ . If  $a \xrightarrow{co} b$ , then the memory access of  $a$  is guaranteed to be completed when  $b$  occurs.*

The term “completed” means in the case of a memory read that the destination memory has been read and in the case of a write that the memory modification is visible at the destination memory. More specifically, for a local buffer access  $b \in B$ , “completed” refers to “local completion”, and for a remote access  $r \in R$ , “completed” refers to “remote completion”.

An essential observation from this definition is that  $a \xrightarrow{co} b$  implies  $a \xrightarrow{hb} b$  because event  $a$  must happen before event  $b$  to have  $a$  completed before  $b$ . The converse does, as discussed, not hold. Therefore,  $\xrightarrow{co}$  is a subset of  $\xrightarrow{hb}$ .

Another weaker order required in the model is the fence order  $\xrightarrow{fo}$  that provides ordering guarantees between remote writes. The fence order  $\xrightarrow{fo}$  is defined as follows:

**Definition 4.3.** *Let  $a \in R^{wr}$  and  $b \in E$  with  $\text{org}(a) = \text{org}(b)$ . If  $a \xrightarrow{fo} b$ , then  $a$  is fence-ordered before  $b$ . That means, if  $b \in R^{wr}$  and  $\text{tgt}(a) = \text{tgt}(b)$ , the write of  $a$  is guaranteed to be completed before the write of  $b$  is completed.*

Compared to the consistency order  $a \xrightarrow{co} b$  which guarantees that  $a$  will be completed when  $b$  occurs, the fence order  $a \xrightarrow{fo} b$  only provides an ordering guarantee between remote writes  $a$  and  $b$  with the same origin and the same target, i.e., when  $b$  completes, then  $a$  must have completed before. The fence order  $\xrightarrow{fo}$  is a subset of the program order  $\xrightarrow{po}$ . In particular, it is used to represent the `shmem_fence` construct that establishes an order between remote writes and avoids data races although  $\xrightarrow{co}$  is not given between conflicting writes. The difference of  $\xrightarrow{co}$  and  $\xrightarrow{fo}$  will be explained in the following subsections in detail.

Table 4.2: Mapping of MPI, SHMEM, and GASPI completion to model completion events. Adapted from [93].

| Event                 | Routines  |
|-----------------------|---|
| collective flush (cf) | MPI_Win_fence, shmem_barrier_all  |
| local flush (lf)      | MPI_Win_flush(_all), MPI_Win_unlock(_all), MPI_Win_flush_local(_all), MPI_Win_complete, shmem_quiet, gaspi_wait |
| remote flush (rf)     | MPI_Win_flush(_all), MPI_Win_unlock(_all), shmem_quiet  |
| target notify (tn)    | SHMEM atomic write calls, shmem_put_signal, gaspi_notify, gaspi_write_notify, MPI_Win_complete                  |
| target wait (tw)      | shmem_(signal_)wait_until, gaspi_notify_waitsome, MPI_Win_wait  |
| fence (fnc)           | shmem_fence, gaspi_notify, MPI_Win_complete   |
| local complete (lc)   | MPI request completion (e.g., MPI_Wait) <sup>1</sup> , gaspi_notify_waitsome <sup>2</sup>                       |

<sup>1</sup> Assuming that the MPI request corresponding to the request-based RMA operation is completed.

<sup>2</sup> gaspi\_notify\_waitsome provides local completion for the associated gaspi\_read\_notify operation.

The completion mechanisms in RMA play a significant role in the definition of consistency. The model distinguishes seven different kinds of completion events: collective flush (cf), local flush (lf), remote flush (rf), target notify (tn), target wait (tw), fence (fnc), and local complete (lc). The concrete completion routines defined in MPI RMA, SHMEM, and GASPI can be mapped to those completion events as shown in Table 4.2. In the following, *CF*, *LF*, *RF*, *TN*, *TW*, *FNC*, and *LC* represent the sets of the respective completion events in the set of all events  $E$ .

Since the term “the memory access is guaranteed to be completed” of the  $\xrightarrow{co}$  and  $\xrightarrow{fo}$  definition is not a priori rigorously specified in the RMA models, the formal model defines *inference rules* to derive  $\xrightarrow{co}$  and  $\xrightarrow{fo}$ . The rules shown in Figure 4.2 model the interaction of memory access events, completion events, and the happened-before relation. As usual, the premises are denoted in the upper half of a rule and the conclusion (the effect on the  $\xrightarrow{co}$  and  $\xrightarrow{fo}$  order) is denoted in the lower half. The system consists of nine rules that are explained in detail in the following sections.

### Transitivity of Consistency

Whenever a memory access in  $a \in M$  is completed before an event  $b \in E$  occurs, it is also completed before any event  $c$  that happened after  $b$ . This transitivity of the consistency order  $\xrightarrow{co}$  with respect to the happened-before order  $\xrightarrow{hb}$  is defined in rule R1. In the following rules, whenever a completion event  $c$  ensures the completion of a memory access  $m \in B \cup R$ , this is modeled as  $m \xrightarrow{co} c$ . Due to the transitivity rule, any event  $e \in E$  with  $c \xrightarrow{hb} e$  will also observe the completion of the event  $m$ , i.e.,  $m \xrightarrow{co} e$ .

|  |   |
|--|---|
| Transitivity of co (R1):<br>$\frac{a \in M \quad b, c \in E \quad a \xrightarrow{co} b \quad b \xrightarrow{hb} c}{a \xrightarrow{co} c}$  | Local memory access (R2):<br>$\frac{m \in L \quad e \in E \quad m \xrightarrow{po} e}{m \xrightarrow{co} e}$                                |
| Collective flush (R3):<br>$\frac{cf \in CF \quad m \in B \cup R \quad \text{tgt}(m) \in \text{grp}(cf) \quad m \xrightarrow{po} cf}{m \xrightarrow{co} cf}$  |   |
| Local flush (R4):<br>$\frac{lf \in LF \quad m \in B \cup R^{rd} \quad \text{tgt}(m) \in \text{grp}(lf) \quad m \xrightarrow{po} lf}{m \xrightarrow{co} lf}$  |   |
| Remote flush (R5):<br>$\frac{rf \in RF \quad m \in R \cup B^{rd} \quad \text{tgt}(m) \in \text{grp}(rf) \quad m \xrightarrow{po} rf}{m \xrightarrow{co} rf}$   |   |
| Fence (R6):<br>$\frac{fnc \in FNC \quad m \in R^{wr} \quad e \in R^{wr} \cup TN \quad \text{tgt}(m) = \text{tgt}(e) \quad m \xrightarrow{po} fnc \quad fnc \xrightarrow{po} e}{m \xrightarrow{fo} e}$            |   |
| Target notify/wait (R7):<br>$\frac{m \in R^{wr} \quad tn \in TN \quad tw \in TW \quad \text{sig}(tn) = \text{sig}(tw) \quad \text{tgt}(tn) = \text{org}(tw) \quad m \xrightarrow{fo} tn}{m \xrightarrow{co} tw}$ |   |
| Remote access of tn is fo (R8):<br>$\frac{tn \in TN \quad m \in R^{wr} \quad \text{access}(tn) = m}{m \xrightarrow{fo} tn}$  | Individual local completion (R9):<br>$\frac{lc \in LC \quad m \in B \cup R^{rd} \quad \text{id}(m) = \text{id}(lc)}{m \xrightarrow{co} lc}$ |

Figure 4.2: Inference rules that describe the consistency order  $\xrightarrow{co}$  and the fence order  $\xrightarrow{fo}$  of RMA models. Adapted from [93].

### Local Memory Accesses

Local memory accesses are unrelated to any RMA operation and assumed to be completed immediately, as shown in rule R2. In other words, the effect of local memory accesses is visible to all events that follow in program order  $\xrightarrow{po}$ . Thus, for local memory accesses, the consistency order  $\xrightarrow{co}$  is identical to the program order  $\xrightarrow{po}$  and across processes to the happened-before order  $\xrightarrow{hb}$ .

### Flushes

Flushes represent all RMA routines that perform bulk completion, i.e., completion of a set of RMA events that happened before the routine. The first variant is the *collective flush* ( $cf$ ) event which is collectively invoked on a designated group of processes  $\text{grp}(cf)$ . It ensures that RMA operations issued before the flush event (in program order) and targeted

to a process in the group  $\text{grp}(cf)$  are completed, both the local buffer accesses and the remote accesses, as shown in rule R3. By that, it models the semantics of *MPI\_Win\_fence* and *shmem\_barrier\_all*. The premise  $\text{tgt}(m) \in \text{grp}(cf)$  means that only RMA operations with a target that is part of the process group are completed. This models the property of communicators in MPI or teams in SHMEM, where only subgroups of processes ensure completion between each other. Together with the transitivity rule R1, it means that all events happening after the collective flush *cf* will observe the memory effect of the completed RMA operations.

The second variant is the *local flush (lf)* event. It provides local completion for all RMA operations addressed to a designated group of target processes. Local completion means that all matching local buffer accesses *B* are completed, as depicted in rule R4. For reading RMA operations, local completion also implies remote completion, so a local flush ensures that all matching remote reads in  $R^{rd}$  are completed. Many different RMA routines provide local flush semantics, e.g., *MPI\_Win\_flush\_local*, *shmem\_quiet*, or *gaspi\_wait*.

The third variant is the *remote flush (rf)* event as a counterpart to the local flush. It provides remote completion for all RMA operations addressed to a designated group of target processes. Remote completion means that all matching remote accesses in *R* are completed, as depicted in rule R5. For writing RMA operations, remote completion also implies local completion, so a remote flush also ensures that all matching local buffer reads in  $B^{rd}$  are completed. Remote flush semantics is provided in *MPI\_Win\_flush*, *MPI\_Win\_unlock*, and *shmem\_quiet*. GASPI has no remote flush functionality.

### Fences

SHMEM provides a routine named *shmem\_fence* that explicitly orders remote write accesses: All remote write operations issued from the same origin to the same target *before* the fence routine will be visible before all remote write operations from the same origin to the same target issued *after* the fence. It does not provide completion but only ordering guarantees, which is modeled by the previously introduced fence order  $\xrightarrow{fo}$  and not to be confused with the consistency order  $\xrightarrow{co}$ . Fence events *fnc* establish an order between remote writes and avoid data races although  $\xrightarrow{co}$  is not given between conflicting writes. The routines *shmem\_fence*, *gaspi\_notify* and *MPI\_Win\_complete* are mapped to a corresponding *fnc* event that ensures  $\xrightarrow{fo}$ , as defined through rule R6.

### Notify-Wait

SHMEM and GASPI provide a notify-wait mechanism to ensure remote completion of RMA operations that is incorporated in the *target notify (tn)* and *target wait (tw)* events. Both events are associated with a signal address captured by the *sig* function. As depicted in rule R7, if a target notify *tn* issued by the origin matches a target wait *tw* issued by the target with the same signal address ( $\text{sig}(tn) = \text{sig}(tw)$ ), then a consistency relation

is established between fence-ordered remote writes  $m$  from the origin to the target wait  $tw$ .

In SHMEM, atomic write calls can be interpreted as  $tn$ , where the remote memory address is the signal address  $\text{sig}(tn)$ , and the corresponding  $\text{shmem\_wait\_until}$  at the target can be interpreted as  $tw$ . All remote write accesses that are fence-ordered before the atomic write call (via  $\text{shmem\_fence}$ ) will be consistent to  $tw$ . The routines  $\text{gaspi\_notify}$  and  $\text{gaspi\_notify\_waitsofme}$  can be mapped similarly to  $tn$  and  $tw$  with the difference that  $\text{gaspi\_notify}$  provides fence-ordering of all previous memory writes implicitly, no explicit call such as  $\text{shmem\_fence}$  is provided in GASPI.

SHMEM additionally provides a routine  $\text{shmem\_put\_signal}$  which combines a notification and a remote write that is implicitly fence-ordered. The same semantics is also provided by  $\text{gaspi\_write\_notify}$ . Both provide *only* a fence order for the associated remote write. Other previously issued remote writes are *not* fence-ordered to those calls. To reflect this property in the model, the  $tn$  event has an additional property  $\text{access}(tn)$  which returns the associated remote write  $m \in R^{wr}$  (if there is one). The inference rule R8 captures the property that this associated remote write is fence-ordered.

Although not a notify-wait mechanism itself, the remote completion behavior of PSCW is also captured by those events: The call to  $\text{MPI\_Win\_complete}$  can be interpreted as both  $fnc$  and target notify  $tn$  and the call to  $\text{MPI\_Win\_wait}$  as target wait  $tw$ . The signal address used in that case is the MPI RMA window object combined with the pair of origin and target process.

### Individual Local Completion

In addition to the flush-based local completion, which completes a set of RMA operations, MPI RMA and GASPI provide mechanisms for fine-granular local completion control. For request-based RMA operations in MPI RMA, the request completion implied by calls such as  $\text{MPI\_Wait}$  provides local completion *only* of the operation associated with that request. The same holds for  $\text{gaspi\_read\_notify}$  operations that can be individually locally completed with  $\text{gaspi\_notify\_waitsofme}$ .

The model captures individual local completion in rule R9. If the associated identifier  $\text{id}(m)$  of the memory access  $m$  matches the identifier  $\text{id}(lc)$  of the local completion  $lc$  event, then local completion is ensured. Request-based operations in MPI RMA and  $\text{gaspi\_read\_notify}$  have such an associated identifier: For MPI RMA, the used identifier is the request object and for GASPI, it is the notification value. SHMEM does not provide individual local completion routines. Instead, there are RMA communication calls that are inherently blocking, i.e., a call to  $\text{shmem\_put}$  or  $\text{shmem\_get}$  is defined to block until local completion is ensured. This property can also be modeled through individual local completion where the locally blocking communication routines are directly mapped to a matching local completion event  $lc$ .

### 4.1.3 Data Race Formalization

After having defined the consistency order  $\xrightarrow{co}$  and the fence order  $\xrightarrow{fo}$  through inference rules in Figure 4.2 and together with the definition of conflicting memory accesses in Definition 4.1, a data race in RMA is defined as follows:

**Definition 4.4** (RMA Data Race, adapted from [93]). *Let  $a, b \in M$  be conflicting memory events. There is a data race between  $a$  and  $b$  if they are neither consistency-ordered nor fence-ordered, i.e., the following two conditions both hold:*

1.  $a \not\xrightarrow{co} b$  and  $b \not\xrightarrow{co} a$ ,
2.  $a \not\xrightarrow{fo} b$  and  $b \not\xrightarrow{fo} a$ .

*If  $a, b \in B \cup L$ , then the race is a local buffer race, otherwise a remote race.*

As previously described, a data race between conflicting memory accesses can either be avoided by establishing consistency  $\xrightarrow{co}$  between events or through a fence order  $\xrightarrow{fo}$  that only affects remote writes from the same origin to the same target. Both properties have to be captured by the data race detection model that is presented in the following section.

## 4.2 RMA Race Detection with Concurrent Regions

As the previous section has shown, detecting data races in RMA models at runtime requires a concept to track the  $\xrightarrow{co}$  and the  $\xrightarrow{fo}$  order at runtime. Since the  $\xrightarrow{co}$  order is a subset of the  $\xrightarrow{hb}$  order which can be accurately tracked by vector clocks as shown in Chapter 3, a clock-based approach to track  $\xrightarrow{co}$  is a natural choice. In particular, for tracking the  $\xrightarrow{co}$  order, a memory access  $m \in B \cup R$  belonging to an RMA operation is mapped to an event tuple  $(a, b)$  during which  $m$  may occur. For tracking the  $\xrightarrow{fo}$  order, each process can use a simple epoch counter to record the order, which will be detailed later on.

According to [93], the *concurrent region* of a memory access  $m \in B \cup R$  is a tuple  $(a, b)$  with  $a, b \in E$  describing the earliest possible point in time represented by  $a$  and the latest possible point in time represented by  $b$  when the underlying memory operation may take place. As previously described, RMA operations consist of a local buffer access event and a remote access event. For the local buffer accesses in  $B$ , there is a *local concurrent region* covering when the actual memory effect of a local buffer access event may occur. For remote accesses in  $R$ , there is (1) a *local concurrent region* covering when the underlying memory access may take place from *the origin's point of view* and (2) a *remote concurrent region* covering when the underlying memory access may take place from *the target's point of view*. Overlap checks on the concurrent regions can be used to detect RMA data races, as discussed in the following sections. The later presented RMA

race detector extends the shared-memory race detector ThreadSanitizer [95] and relies on the computation of those concurrent regions.

### 4.2.1 Local Buffer Races

Any local buffer access event  $b \in B$  of an RMA operation spans up a local concurrent region  $LCR(b)$ : The earliest point when its memory access may occur is when the event  $b$  is issued itself, and the latest point is the next completion event  $c \in E$  in program order  $\xrightarrow{po}$  that provides local completion. Formally, the earliest matching completion event  $c$  is defined as

$$b \xrightarrow{co} c \wedge \nexists c' \in E : c' \xrightarrow{po} c \wedge b \xrightarrow{co} c'.$$

The resulting local concurrent region is  $LCR(b) := (b, c)$ . For local concurrent regions, any memory event  $m \in M$  happening in between  $b$  and  $c$  is not consistent with  $b$ , i.e.,

$$\forall m \in M : b \xrightarrow{hb} m \xrightarrow{hb} c \implies m \xrightarrow{cq} b \wedge b \xrightarrow{cq} m.$$

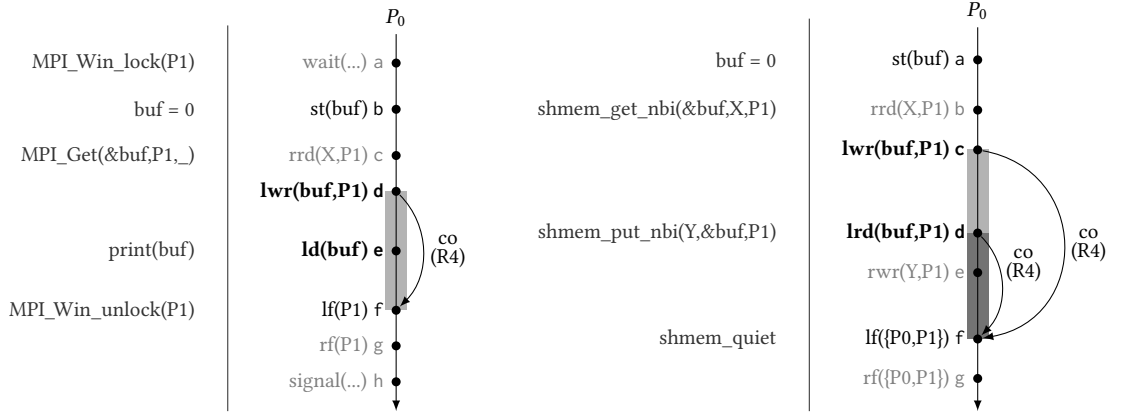
Since the fence order  $\xrightarrow{fo}$  is irrelevant for local buffer races because it only orders remote writes, checking for overlapping concurrent regions is enough to detect local buffer races.

Figure 4.3a shows an example of an overlapping *MPI\_Get* access with a conflicting *print* call. The different observed MPI calls and memory accesses are translated into corresponding events in the consistency and happened-before model. *MPI\_Get* is translated into a remote read ( $c = rrd$ ) and a local buffer write ( $d = lwr$ ). The memory write of  $d = lwr$  occurs at earliest with the event itself and at latest with the local flush in event  $f = lf$  implied from the *MPI\_Win\_unlock*, as described by the consistency rule R4. Therefore, the concurrent region is  $LCR(d) = (d, f)$  which overlaps with the read access of *print(buf)* in event  $e$ , implying that  $d \xrightarrow{cq} e$  and  $e \xrightarrow{cq} d$ , leading to a local buffer race.

Figure 4.3b shows an example with an overlapping *shmem\_get\_nbi* and *shmem\_put\_nbi* access. The *shmem\_get\_nbi* access initiates a local buffer write in  $c = lwr$  and the *shmem\_put\_nbi* access a local buffer read in  $d = lrd$ , both completed with a *shmem\_quiet* ( $f$ ), again as implied by consistency rule R4. Here,  $c$  is conflicting with  $d$  and the local concurrent region  $LCR(c) = (c, f)$  is overlapping with  $LCR(d) = (d, f)$ , implying that  $c \xrightarrow{cq} d$  and  $d \xrightarrow{cq} c$ , leading to a local buffer race. Similar scenarios, as shown in the two examples, can be constructed using the collective flush *cf* and individual local completion *lc* events.

Detecting local buffer races only requires observing and storing the concurrent region information: An on-the-fly algorithm to detect local buffer races may record the local buffer access events and store all current non-completed local buffer access events. Whenever a memory access occurs, it has to be checked with all non-completed local buffer access events for a conflict. If there is a conflict, then a race is reported.

## 4 Event-Based RMA Race Detection Model



- (a) The concurrent region of the local buffer write (d) overlaps the local load of the print (e), leading to a data race. (b) The concurrent region of the local buffer write (c) overlaps the concurrent region of the other local buffer write (d), leading to a data race.

Figure 4.3: Concurrent region examples of local buffer races. The conflicting events are printed in bold and the concurrent regions are printed as gray-shaded areas. Adapted from [93].

### 4.2.2 Remote Races

For remote accesses, the detection of concurrent regions is more involved because it requires considering the synchronization of the origin with the target process. Unlike a local buffer access, a remote access  $r \in R$  is issued *at the origin* and its underlying memory access takes place *at the target*. From the origin perspective, the memory effect of a remote access  $r \in R$  occurs at earliest with the event  $r$  itself and at latest with the earliest next matching remote completion event  $c$  which is characterized with

$$r \xrightarrow{co} c \wedge \nexists c' \in E : c' \xrightarrow{po} c \wedge r \xrightarrow{co} c'.$$

This tuple  $LCR(r) = (r, c)$  defines the local concurrent region of  $r$ , i.e., from the origin's point of view. For race detection, it is also required to understand when  $r$  might occur from the target's point of view. For any remote access  $r \in R$  issued on an origin process  $P_o$  to a target process  $P_t$ , the *remote concurrent region*  $RCR(r, c)$  is the tuple  $(m, n)$ , where  $m \in E_t$  is the earliest point and  $n \in E_t$  is the latest point from the target's point of view at which the memory effect of  $r$  might take place. In other words, it is the local concurrent region  $LCR(r)$  projected to the target's point of view. An example of the projection of  $LCR$  to  $RCR$  is shown in Figure 4.4.

There are two ways of checking the  $\xrightarrow{co}$  order for remote accesses from an algorithmic perspective: (1) Checking for overlapping local concurrent regions  $LCR$  of (conflicting) accesses by comparing the full vector clocks of the events from the origin side and (2) checking for overlapping remote concurrent regions  $RCR$  at the target. The latter one has the benefit that it allows for an efficient implementation and perfectly fits to the fiber concept [45] of ThreadSanitizer [95] that is used in the implementation described

in Chapter 5. Comparing the remote concurrent regions at the target, however, has the drawback that overlapping remote concurrent regions are a necessary but not a sufficient condition for an RMA data race. As will be discussed later by an example, the local concurrent regions still have to be compared to avoid false positives. The following sections precisely describe how the remote concurrent region  $RCR$  can be determined for race detection.

The first step in determining  $RCR$  is to characterize when an event from the origin may occur from the target's point of view. Formally, the mapping from the origin's point of view to the target's point of view can be analyzed with the happened-before order  $\xrightarrow{hb}$  as *last signal* and *next wait*:

**Definition 4.5** (Last Signal, adapted from [90]). *Let  $a \in E_o$  be an event at  $P_o$ . The last signal of  $P_t$  to  $P_o$  before  $a$ , denoted as  $LS_{t \rightarrow o}(a)$ , is the event  $b \in E_t$  where*

- (1)  $b \xrightarrow{hb} a$ ,
- (2)  $\nexists b' \in E_t : b \xrightarrow{hb} b' \xrightarrow{hb} a$ .

Intuitively, the last signal of  $P_t$  before  $a$  is the latest event  $b$  that happened at  $P_t$  before  $a$ . As vector clocks fully capture the  $\xrightarrow{hb}$  order, the last signal can be determined through vector clocks by looking at the corresponding entry of the vector clock of process  $P_t$ :

**Theorem 4.1** (adapted from [90]). *Let  $a \in E_o$  be an event at  $P_o$ . The last signal  $LS_{t \rightarrow o}(a)$  is the event  $b \in E_t$  where  $V(a)[t] = V(b)[t]$ .*

This immediately follows out of the vector clock exchange in Definition 3.5: Let  $a \in E_o$  and  $b \in E_t$  with  $V(a)[t] = V(b)[t]$ . This means that  $P_o$  must have (transitively) merged  $V(b)$  at latest in event  $a$ . But this also means that  $V(b)[o] < V(a)[o]$  and thus  $V(b) < V(a)$  implying that  $b \xrightarrow{hb} a$ . Also, there cannot exist  $b' \in E_t$  with  $b \xrightarrow{hb} b' \xrightarrow{hb} a$ : Assuming that such an event  $b' \in E_t$  exists, this would require that  $V(b)[t] < V(b')[t] \leq V(a)[t]$  which cannot be the case since  $V(a)[t] = V(b)[t]$ . Thus, such a  $b'$  cannot exist, and there is a unique  $b$  for which  $b = LS_{t \rightarrow o}(a)$  holds.

The next wait is the counterpart to the last signal which determines for an event  $a$  at  $P_o$  the earliest event  $b$  happening after  $a$  at  $P_t$ :

**Definition 4.6** (Next Wait, adapted from [90]). *Let  $a \in E_o$  be an event at  $P_o$ . The next wait of  $P_t$  for  $P_o$  after  $a$ , denoted as  $NW_{t \rightarrow o}(a)$ , is the event  $b \in E_t$  where*

- (1)  $a \xrightarrow{hb} b$ ,
- (2)  $\nexists b' \in E_t : a \xrightarrow{hb} b' \xrightarrow{hb} b$ .

Intuitively, the next wait of  $P_t$  after  $a$  is the earliest event  $b$  that happened at  $P_t$  after  $a$ . Analogously to the last signal, the next wait can be determined through vector clocks:

**Theorem 4.2** (adapted from [90]). *Let  $a \in E_o$  be an event at  $P_o$ . The next wait  $NW_{t \rightarrow o}(a)$  is the event  $b \in E_t$  where*

- (1)  $V(b)[o] \geq V(a)[o]$ , and
- (2)  $V(b)[t] \leq V(b')[t] \forall b' \in E_t$  with  $V(b')[o] \geq V(a)[o]$ .

This also follows out of the vector clock exchange in Definition 3.5: Let  $a \in E_o$  and  $b \in E_t$  fulfill the conditions (1) and (2). Due to (1)  $V(b)[o] \geq V(a)[o]$ ,  $P_t$  must have (transitively) merged  $V(a)$  at latest in event  $b$ . But this also means that  $V(b)[t] > V(a)[t]$  implying that  $V(b) > V(a)$  and thus  $a \xrightarrow{hb} b$ . Also, there cannot exist  $b' \in E_t$  with  $a \xrightarrow{hb} b' \xrightarrow{hb} b$ : Assume that such an event  $b' \in E_t$  exists. Applying the vector clock property yields  $V(a) < V(b') < V(b)$ . In particular, it must be  $V(b')[o] \geq V(a)[o]$  and since  $b', b \in E_t$  are issued from the same process, it must be  $V(b')[t] < V(b)[t]$ , but this violates condition (2). Thus, such a  $b'$  cannot exist, and further,  $b$  must be unique, so  $b = NW_{t \rightarrow o}(a)$  holds.

If processes did not synchronize at all, for example, at the beginning of an exchange, i.e., if  $V(a)[t] = 0$ , then the last signal of  $a$  can be assumed to be a dummy event  $b \in P_t$  at the beginning of the execution of  $P_t$ . Similarly, if there is no such event  $b$  with  $V(b)[o] > V(a)[o]$ , then the next wait of  $a$  can be interpreted as the latest event at  $P_t$ .

Having a remote access event  $r \in R \cap E_o$  and the next matching remote completion event  $c \in E_o$ , the remote concurrent region is

$$RCR(r, c) := (LS_{t \rightarrow o}(r), NW_{t \rightarrow o}(c)).$$

In the following, different examples of determining the remote concurrent region will be discussed and illustrated.

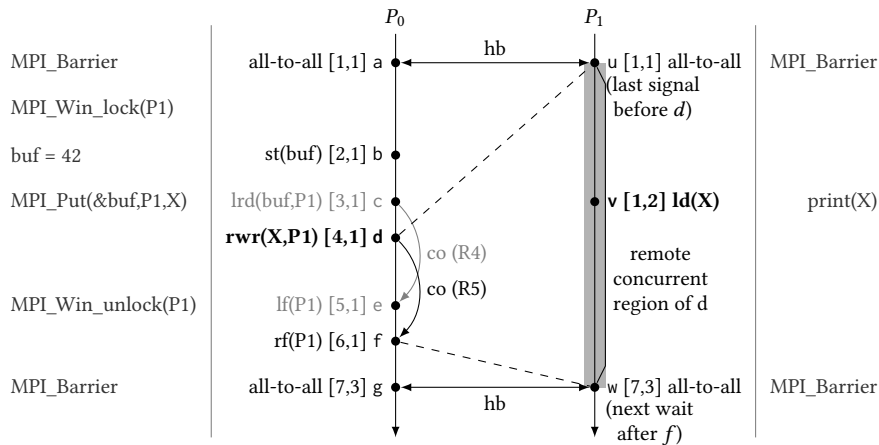


Figure 4.4: Remote concurrent region example of Figure 2.8a. The conflicting events leading to the race are printed in bold. The gray-shaded area is the remote concurrent region of  $d$ . Adapted from [93].

## 4.2 RMA Race Detection with Concurrent Regions

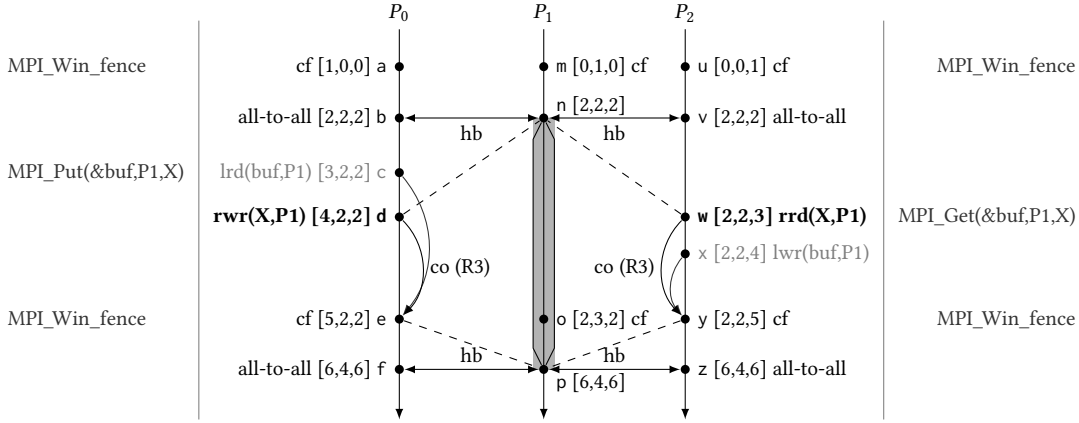


Figure 4.5: Remote concurrent region example using *MPI\_Win\_fence* with two conflicting remote accesses from different origins. The conflicting events leading to the race are printed in bold.  $P_1$  also executes the *MPI\_Win\_fence* statements, which are not depicted here for illustration purposes.

### Remote Flush Example

Figure 4.4 shows how to determine the remote concurrent region of an *MPI\_Put* completed with *MPI\_Win\_unlock*. The remote access starts with  $d = rwr$  and ends with  $f = rf$  from  $P_0$ 's perspective, according to consistency rule R5, so  $LCR(d) = (d, f)$ . The last signal before  $d$  at  $P_1$  is the *all-to-all* synchronization in event  $u$ . This can be determined by looking at the vector clock entry  $V(d)[1] = 1$  and finding the corresponding event at  $P_1$  with the same entry which is event  $u$  with  $V(u)[1] = 1$ . The next wait after  $f$  at  $P_1$  is the *all-to-all* synchronization in event  $w$ . This can be determined by looking at the vector clock entry  $V(f)[0] = 6$  and finding the earliest event  $x \in E_1$  with  $V(x)[0] \geq V(f)[0]$ . In this case, it is event  $w$  with  $V(w)[0] = 7$ . Thus, the tuple  $(d, f)$  from  $P_0$ 's perspective is mapped to the tuple  $RCR(d, f) = (u, w)$  at  $P_1$ 's perspective, as illustrated with the gray-shaded area in Figure 4.4. The conflicting event  $v = ld(X)$  overlaps with the remote concurrent region  $(u, w)$ , indicating that  $d \xrightarrow{co} v$  and  $v \xrightarrow{co} d$ . Since the accesses cannot be  $\xrightarrow{fo}$ -ordered, this example has a remote race.

### Collective Flush Example

Figure 4.5 illustrates a remote race example using *MPI\_Win\_fence* with three processes. Each call to *MPI\_Win\_fence* is represented as collective flush *cf* and a subsequent *all-to-all* event in the model. The *MPI\_Put* at  $P_0$  writing to  $P_1$  and the *MPI\_Get* at  $P_2$  reading from  $P_1$  are conflicting since they both access the same target memory location  $X$  at  $P_1$ . Both operations are completed with the the call to *MPI\_Win\_fence* in events  $e$  and  $y$ , respectively, according to consistency rule R3, so  $LCR(d) = (d, e)$  and  $LCR(w) = (w, y)$ . The remote concurrent region for  $d = rwr$  and  $w = rrd$  is  $RCR(d, e) = (n, p)$  and  $RCR(w, y) = (n, p)$ ,

#### 4 Event-Based RMA Race Detection Model

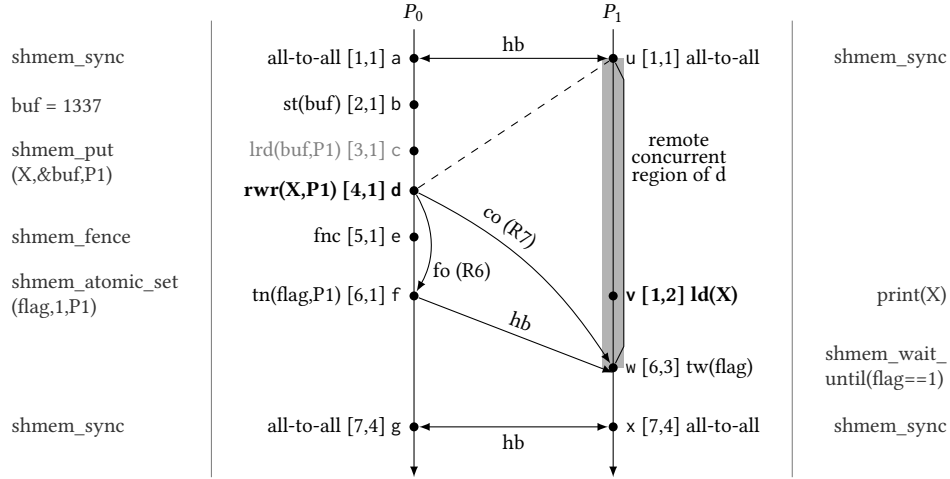


Figure 4.6: Remote concurrent region example of Figure 2.10a. The conflicting events leading to the race are printed in bold. The immediate local completion of *shmem\_put* is not depicted for illustration purposes. Adapted from [93].

because the last signal from target  $P_1$  is event  $n$  and the next wait of  $P_1$  is event  $p$  in both cases. Since those regions are obviously overlapping, it is  $d \xrightarrow{co} w$  and  $w \xrightarrow{co} d$ . Since accesses from different processes can never be  $\xrightarrow{fo}$ -ordered, there is a remote race in the given event trace.

#### Notify-Wait Example

Figure 4.6 shows a remote concurrent region finished with notify-wait completion. The call to *shmem\_put* is mapped to  $d = rwr$  which is  $\xrightarrow{fo}$ -ordered to the notification  $f = tn$  due to the *shmem\_fence* call (rule R6). The connection of  $f = tn$  with  $w = tw$  establishes a  $\xrightarrow{hb}$ -order and also ensures consistency of  $d = rwr$  due to rule R7. The remote concurrent region for  $d = rwr$  is correspondingly  $RCR(d, f) = (u, w)$  and the conflicting  $v = ld(X)$  is within the concurrent region leading to a data race due to missing  $\xrightarrow{co}$ -order between those events (and missing  $\xrightarrow{fo}$ -order).

In notify-wait completion, no remote completion is established at the origin process. As Figure 4.6 shows, there is no event at  $P_0$  that ensures  $\xrightarrow{co}$  of  $d = rwr$  from the origin's perspective. In other words, the origin does not know when  $d = rwr$  will be visible at the target. To still have a defined local concurrent region, the assumption in this case is that the event  $f = tn$  terminates the  $\xrightarrow{fo}$ -ordered write access of  $d$ , so the local concurrent region is  $(d, f)$ . This is, in particular, required to have defined behavior when checking for overlapping local concurrent regions of remote accesses that will be discussed in Section 4.2.3.

The notify-wait example in Figure 4.6 also does not provide completion of the local buffer read  $c = lrd$ , so the local concurrent region of the buffer access is still not terminated

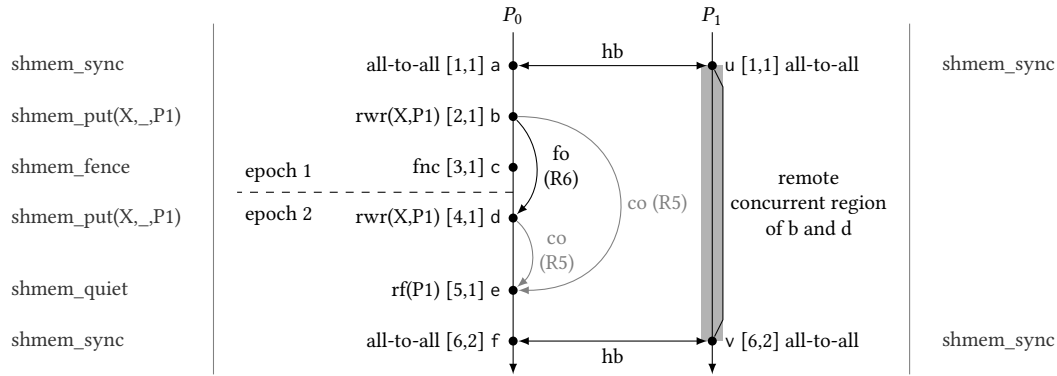


Figure 4.7: Identical remote concurrent regions of two *shmem\_put* calls that are  $f_o$ -ordered and therefore race-free. An epoch counter for fences can be used to detect the ordering between *c* and *e*. The *lrd* events of the *shmem\_put* calls are omitted for illustration purposes. Adapted from [93].

with *shmem\_sync* according to the concurrent region model. However, the event  $w = tw$  ensures consistency of  $d = rwr$  which implies that at this point in the execution of  $P_1$ , the local buffer read of  $c = lrd$  must also be completed. This completion information would then be propagated with *shmem\_sync* to  $P_0$ . Capturing this information with the given concurrent region construct would be involved as additional information on the consistency of the events would have to be propagated between the processes to know that the synchronization of *shmem\_sync* implicitly provides local completion. Further, a program relying on implicit local completion effects combined with *shmem\_sync* is an unrealistic use case because local completion in this case is still usually enforced, e.g., with *shmem\_quiet* or *shmem\_barrier* in the case of SHMEM. Instead, a typical situation where such implicit local completion matters are bidirectional notifications between processes, i.e.,  $P_0$  performs a remote write and signals to  $P_1$  which in turn signals back to  $P_0$  and implicitly provides local completion of  $P_0$ 's remote write. To cover such cases, the race detection implementation assumes that a matching  $tw$  event at a process  $P_0$  waiting for the signal of process  $P_1$  ensures completion of local buffer accesses at  $P_0$  associated with remote writes to  $P_1$ . An example with details on this implementation decision is further discussed in Section 6.1.3.

### Fence Example

Figure 4.7 shows a model example of two *shmem\_put* operations issued from the same origin  $P_0$  to the same memory location  $X$  at the same target  $P_1$ , ordered by a *shmem\_fence*. Both *rwr* events *b* and *d* are conflicting. Since both are completed with the  $e = rf$  event, the remote concurrent region of both events is  $(u, v)$ . This suggests that  $b \xrightarrow{co} d$  and  $d \xrightarrow{co} b$ . However, the event  $c = fnc$  ensures that  $b \xrightarrow{fo} d$ , according to consistency rule R6. Thus, the memory effect of *b* is guaranteed to be visible before *d* and there is no race according to Definition 4.4.

#### 4 Event-Based RMA Race Detection Model

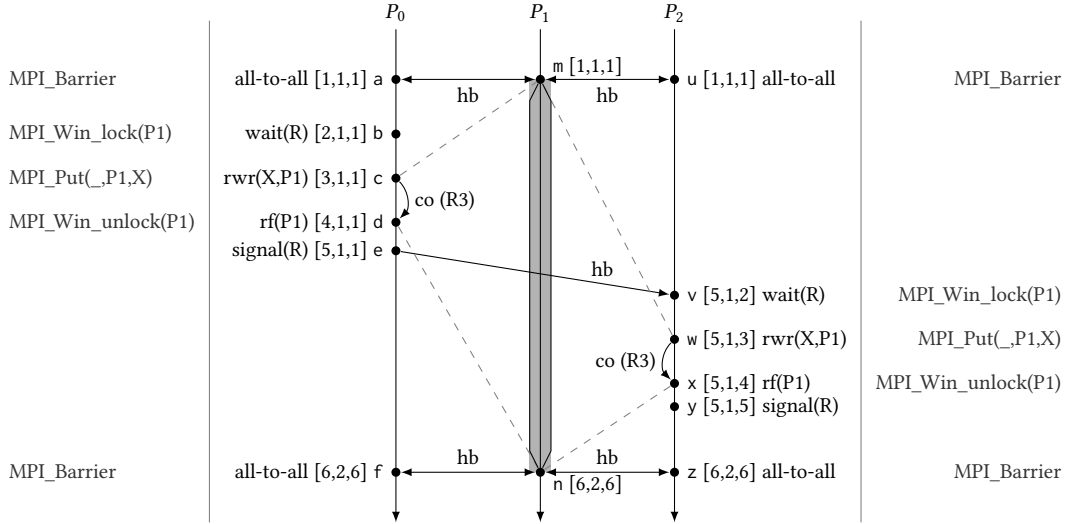


Figure 4.8: External synchronization of two remote writes (event  $c$  and event  $w$ ) that have the same remote concurrent region at the target. There is no data race in this example. The  $lrd$  events of the  $MPI\_Put$  calls are omitted for illustration purposes.

This example shows that checking for overlapping remote concurrent regions is insufficient to detect remote races, because the  $\xrightarrow{fo}$ -order must also be considered. Since  $\xrightarrow{fo}$  is a concept that only establishes the ordering of remote writes of the *same* origin, it is enough to check if there is a  $fnc$  event between conflicting writes. For an on-the-fly detection, a simple solution is to use an epoch counter that is incremented on every  $fnc$  event. Whenever there are conflicting writes from the same origin to the same target, different epoch counters indicate an  $\xrightarrow{fo}$ -order between those events.

#### External Synchronization Example

The remote concurrent region is insufficient to capture  $\xrightarrow{co}$  in all cases, as Figure 4.8 shows.  $P_0$  and  $P_2$  both perform a remote write to the same location at  $P_1$ , but the access is coordinated between the processes using MPI RMA locks. According to the consistency model, the remote write  $c = rwr$  issued at  $P_0$  is completed in  $d = rf$ , so  $LCR(c) = (c, d)$ . The remote write  $w = rwr$  issued at  $P_2$  is completed in  $x = rf$ , so  $LCR(w) = (w, x)$ . The local concurrent regions  $LCR(c)$  and  $LCR(w)$  are not overlapping because there is  $\xrightarrow{hb}$  synchronization ensuring that  $d \xrightarrow{hb} w$  which implies that  $c \xrightarrow{co} w$  according to rule R1. The target  $P_1$  is entirely unaware of the external synchronization between  $P_0$  and  $P_2$ , which avoids the data race between the accesses. Correspondingly, the remote concurrent region of  $c$  and  $w$  is  $RCCR(c, d) = RCCR(w, x) = (m, n)$  for both accesses which would indicate that there is no consistency established between those accesses.

To detect the  $\xrightarrow{co}$ -order between the accesses in this particular case, the vector clock ranges of the local concurrent regions, i.e., the *origin* perspective, have to be *additionally*

compared. If they do not overlap in terms of the  $\xrightarrow{hb}$ -order, then there is no data race. Comparing the corresponding vector clock tuples of  $LCR(c)$  and  $LCR(w)$ , i.e.,  $(V(c), V(d))$  and  $(V(w), V(x))$ , shows that they are disjoint due to  $V(d) = [4, 1, 1] < [5, 1, 3] = V(w)$ , confirming that  $c \xrightarrow{co} w$  and there is no race in this example.

### 4.2.3 Race Detection Algorithm

As the previous examples have shown, detecting local buffer races and remote races with vector clocks involves multiple checks. The algorithm depicted in Figure 4.9 summarizes the race detection scheme discussed for the different examples and is suited for on-the-fly detection. The input assumes two local concurrent regions, i.e., in the form of  $(m, c) \in M \times E$  where  $m$  and  $c$  have the following meanings: (1) If  $m$  is a local buffer access  $m \in B$ , then  $c$  is the next event in program order that ensures local completion. (2) If  $m$  is a remote access  $m \in R$ , then  $c$  is the next event in program order that ensures remote completion. (3) If  $m$  is a local memory access  $m \in L$ , then it is immediately completed and  $c$  is the memory access itself, i.e.,  $m = c$ .

For the two local concurrent regions  $(m_1, c_1)$  and  $(m_2, c_2)$ , the algorithm returns false if there is no race between the accesses and true if there is a race. First, it checks according to Definition 4.1 if the two memory accesses  $m_1$  and  $m_2$  are conflicting (lines 2 – 4). If so, it continues to check if the concurrent regions at the destination are disjoint (lines 6 – 11). For that, a distinction between local (buffer) accesses and remote accesses has to be made: If a memory access  $m$  is a remote access, i.e.,  $m \in R$ , then the local concurrent region is translated to its remote concurrent region first. Otherwise, the local concurrent region is not translated. This is shown in the procedure *GetDestCR* (lines 25 – 29). The overlap check is then just a comparison of a single vector clock value as shown in line 10, namely the clock value at the destination process, since all concurrent regions are mapped to the perspective of the same (destination) process. If the concurrent regions do not overlap, there is no race, and the algorithm can return false.

If both  $m_1$  and  $m_2$  are remote accesses, two extra checks have to be performed as previously described with the fence example and the external synchronization example in Section 4.2. The first check verifies if the accesses are  $\xrightarrow{fo}$ -ordered which requires that both accesses are write accesses, from the same origin, and their epochs (which are counted locally on every *fn*c event) are not equal. The second check verifies whether external synchronization between accesses ensures consistency even though the remote concurrent regions of the accesses overlap. As previously described in the example, it requires comparing the full vector clocks of the local concurrent regions, as shown in line 19.

The algorithm's checks are ordered so that it returns as early as possible if there is no race between the accesses. In practice, nearly all of the analyzed access pairs in the algorithm do not race because the amount of data races in a program is typically low (or in a correct program, there is no race at all). The check for conflicting  $m_1$  and  $m_2$  is cheap as it requires comparing the accessed address and access type. Also, the concurrent region

#### 4 Event-Based RMA Race Detection Model

**Require:**  $LCR(m_1) = (m_1, c_1) \in M \times E$ ,  $LCR(m_2) = (m_2, c_2) \in M \times E$

```

1: procedure CHECKRMARACE( $m_1, c_1, m_2, c_2$ )
2:    $\triangleright$  Check if accesses are conflicting (same destination, memory address, atomicity)
3:   if not IsConflicting( $m_1, m_2$ ) then
4:      $\sqsubset$  return False
5:
6:    $d := dst(m_1)$ 
7:    $(m'_1, c'_1) := GetDestCR(m_1, c_1)$ 
8:    $(m'_2, c'_2) := GetDestCR(m_2, c_2)$ 
9:    $\triangleright$  Check if concurrent regions at the destination are disjoint (i.e., not overlapping)
10:  if  $V(m'_2)[d] > V(c'_1)[d]$  or  $V(m'_1)[d] > V(c'_2)[d]$  then
11:     $\sqsubset$  return False
12:
13:   $\triangleright$  Additional checks if both  $m_1$  and  $m_2$  are remote accesses
14:  if  $m_1 \in R$  and  $m_2 \in R$  then
15:     $\triangleright$  Check if accesses are fence-ordered
16:    if  $m_1 \in R^{wr}$  and  $m_2 \in R^{wr}$  and  $org(m_1) = org(m_2)$  and  $epoch(m_1) \neq epoch(m_2)$  then
17:       $\sqsubset$  return False
18:     $\triangleright$  Check if vector clock intervals of local concurrent regions are disjoint (i.e., not overlapping)
19:    if  $V(m_2) > V(c_1)$  or  $V(m_1) > V(c_2)$  then
20:       $\sqsubset$  return False
21:
22:   $\triangleright$  Neither consistency- nor fence-ordered, so there is a race
23:   $\sqsubset$  return True
24:
25: procedure GETDESTCR( $m, c$ )
26:   if  $m \in L \cup B$  then
27:      $\sqsubset$  return ( $m, c$ )
28:   else if  $m \in R$  then
29:      $\sqsubset$  return RCR( $m, c$ )

```

Figure 4.9: Race checking algorithm for races in RMA. The inputs are the local concurrent regions of the memory accesses  $m_1$  and  $m_2$ . The algorithm returns “true” if  $m_1$  and  $m_2$  lead to a data race, otherwise false. The algorithm checks if the accesses are conflicting, then it performs an overlap check of the concurrent regions at the destination. If both  $m_1$  and  $m_2$  are remote accesses, two additional checks (fence order and local concurrent region overlap) are performed.

check at the destination only requires the mapping to the remote concurrent region and then two comparisons of single vector clock entries. If both accesses are remote accesses, then the additional checks involve the fence order verification and an expensive check of the full vector clocks, which is why it is performed at the end.

The algorithm returns true for two memory accesses  $m_1, m_2 \in M$  if and only if they are (1) conflicting, (2) not  $\xrightarrow{co}$ -ordered, and (3) not  $\xrightarrow{fo}$ -ordered, so according to Definition 4.4, it can detect races in the model accurately with the exceptions discussed in the next section. Chapter 5 discusses how this algorithm is used in an on-the-fly data race detector for RMA programs.

## 4.3 Limitations

The presented RMA race detection model is an abstraction of the semantics of RMA programs. Although it captures most of the semantics, each programming model has specifics that the race detection model does not capture. The model overapproximates some completion behavior which may lead to false negatives (non-detected races). On the other hand, any missed synchronization between processes or completion may lead to false positives (falsely detected data races). Further, hybrid parallelism with multithreading is currently not supported. In the following, those limitations will be discussed in detail.

### 4.3.1 Model-Specific Concepts

All three RMA models, MPI RMA, SHMEM, and GASPI, represent remotely accessible regions in different ways. In MPI RMA and GASPI, remotely accessible regions are represented with specific handles named windows and segments, respectively. Thus, a call such as *MPI\_Put* writes to a memory address relative to the base address of an allocated window. In SHMEM, remote memory regions are directly addressed using absolute memory addresses. The presented model does not capture those different representations of remote memory regions. Instead, it assumes that the accessed memory locations are provided as absolute addresses. In the implementation discussed in Chapter 5, an appropriate translation is performed by model-specific wrapper modules.

The RMA consistency model assumes that bulk-completion events, e.g., collective, local, or remote flush, affect *all* previously issued RMA operations, but this is not always true in practice. In MPI RMA, completion calls *MPI\_Win\_flush\_all(win)* ensure completion of all RMA operations addressed to the window *win*, but not completion of RMA operations addressed to other windows. Similarly, in SHMEM, RMA operations can be associated with *contexts* where corresponding completion calls such as *shmem\_ctx\_quiet* ensure only completion of RMA operations with that *context*. To keep the theoretical model simple, this information is not encoded in the primitives and inference rules. In the implementation described in Chapter 5, a model-agnostic context ID is used to still keep the information on windows in MPI RMA and contexts in SHMEM.

As discussed in Section 4.2, the notify-wait mechanism may provide local completion implicitly through transitive synchronization from the target side. As capturing this information would be involved and is only relevant in rare cases, it is assumed as an overapproximation that a matching *tw* locally completes the associated local buffer access. This avoids falsely detected local buffer races, but might lead to false negatives, as later discussed in Chapter 6.

The PSCW model in MPI RMA is modeled as notify-wait synchronization in the RMA consistency model. While this is sufficient in most cases, this does not capture the property of exposure epochs relevant for PSCW. By using exposure epochs, the target

itself can control with *MPI\_Win\_post* when a remote access of the origin may take place earliest. Thus, with PSCW, the begin of the concurrent region can be delayed to when the target opens the exposure epoch. However, according to the RMA consistency model, the remote concurrent region begins with the last signal from the target with respect to the corresponding remote memory access event, not with the beginning of the exposure epoch. Therefore, the RMA consistency model might determine a larger concurrent region than actually present in the execution, potentially leading to falsely detected races (false positives). MPI RMA is the only programming model that comes with such a concept. Still, to avoid such false positives, a solution is to assume that the origin process calling *MPI\_Win\_start* blocks until the target process calls the matching *MPI\_Win\_post*, i.e., that this pair of calls is synchronizing. This assumption is in line with the MPI standard that allows *MPI\_Win\_start* to block for a matching *MPI\_Win\_post*, but this is not mandated. Then, *MPI\_Win\_post* would be interpreted as the last signal and it would be considered as the start of the concurrent region. This overapproximating assumption of synchronization might, in principle, lead to missed races (false negatives), but only in exotic cases. In particular, a test with state-of-the-art MPI implementations (MPICH<sup>1</sup>, OpenMPI<sup>2</sup>) has shown that *MPI\_Win\_start* always blocks waiting for a matching *MPI\_Win\_post*. Thus, the race detection implementation discussed in Chapter 5 uses this overapproximation of synchronization for PSCW to avoid false positives.

### 4.3.2 Separate Memory Model in MPI RMA

As detailed in Section 2.2.1, MPI supports a separate memory model, assuming that an MPI window consists of a private and public copy. Remote accesses modify the public copy, and local memory accesses modify the private copy. Those copies must be synchronized explicitly using the routine *MPI\_Win\_sync*. The introduced consistency model in this thesis does not support the separate memory model, as it is not of practical relevance in today's systems and is not supported at all by SHMEM and GASPI. However, extending the consistency model to support MPI RMA's separate memory model would be straightforward: For every remote access  $r \in R$  completed by  $c \in E$ , its remote concurrent region  $RCR(r,c)$  would not end with the next wait, but only with the next *MPI\_Win\_sync* routine called at the target after the next wait was observed.

### 4.3.3 Hybrid Parallelism

As for the vector clock exchange model, the consistency and race detection models presented in this chapter assume a single-threaded execution. Multithreading makes the modeling of the consistency semantics in RMA significantly more complex. In particular, the notion of a concurrent region would have to be re-defined to consider synchronization

---

<sup>1</sup><https://www.mpich.org>

<sup>2</sup><https://www.open-mpi.org>

between threads. For that, a hybrid vector clock as proposed in Section 3.6 would be required to determine the last signal and the next wait between the individual threads of the processes.

Further, for all three RMA models, completion routines such as *MPI\_Win\_flush* apply on process-scope, i.e., a thread that calls a completion routine also completes RMA operations previously issued by other threads. Thus, whenever a thread calls a completion routine, the synchronization between threads would have to be analyzed with vector clocks to find those RMA operations of other threads that are guaranteed to occur before that completion call. On the other hand, in future RMA models, there might be completion routines providing completion guarantees on thread-scope as proposed for MPI RMA by Schuchart et al. [84]. This distinction between process-scope and thread-scope completion would further complicate the reasoning on RMA operation completion.

Even without considering thread synchronization, the model can still analyze multi-threaded executions by projecting the events on a single event stream, e.g., if thread 1 observes event *a* and thread 2 concurrently observes event *b*, then both events are projected to a single execution stream where either *a* is ordered before *b* or *b* is ordered before *a* in program order. This serialization, however, means that concurrent events from different threads appear to be ordered when they are not. Further, the synchronization between one thread of a process with a thread of another process would lead to the assumption that all threads of the processes synchronized with each other. In other words, ignoring the details of multithreading would lead to an overapproximation of the  $\xrightarrow{hb}$  and the  $\xrightarrow{co}$ -order. This overapproximation leads to false negatives in the race detection, as the evaluation in Chapter 6 will also show.

## 4.4 Generalizability

The focus on MPI RMA, SHMEM, and GASPI raises the question whether the race detection model also applies to other RMA programming models. Since the model primitives are designed to be very generic, every high-level or low-level RMA programming model should, in principle, match those primitives, which are discussed in the following.

UPC [31] is a C language extension providing a PGAS abstraction with shared arrays between processes, but also procedures such as *upc\_memget* and *upc\_memput*. All those accesses can be interpreted as remote read and write events in the model. Completion and synchronization are ensured with barriers, fences, and locks, which are all concepts that can be analyzed with the race detection model. Its successor UPC++ [4] is a header-only library that provides asynchronous communication operations *rget* and *rput* that can be completed individually by waiting on futures. This is similar to request-based operations in MPI RMA. UPC++ has no other means of completion; it solely relies on the individual completion of operations which is also supported by the model.

Higher-level PGAS models such as Coarray Fortran [69, 70] also boil down to remote read and write operations and completion events hidden behind language constructs. In Coarray Fortran, accesses to a coarray located at a remote process are the remote read and write events. Synchronization and completion are encoded in a *SYNC ALL* and *SYNC IMAGES* construct that also matches the consistency and race detection model.

Lower-level communication libraries also provide primitives with similar completion semantics. UCX<sup>3</sup> is a communication middleware framework used in OpenMPI<sup>4</sup> and Sandia SHMEM<sup>5</sup> to implement RMA functionality. UCX itself specifies RMA routines such as *ucp\_put* and *ucp\_get* for remote accesses, *ucp\_flush* and *ucp\_fence* for consistency, and *ucp\_signal* and *ucp\_wait* for synchronization and translates them to lower-level network protocols. Those routines also exactly match the primitives defined in the consistency and race detection model. Similarly, libfabric<sup>6</sup> as an alternative communication middleware also provides such primitives.

The examples of higher-level and lower-level communication libraries show that the presented race detection model can also be applied to programming models targeting other levels in HPC communication stacks. The remaining challenge is wrapping the concrete library routines to the abstract primitives on the implementation side, as discussed in detail for MPI RMA, SHMEM, and GASPI in Chapter 5.

## 4.5 Related Work

Other RMA modeling approaches and race detection models have been discussed previously in the literature. In the following, those will be summarized and compared to the approach in this thesis.

### 4.5.1 MPI RMA Axiomatic Model

Hoefler et al. [38] model the consistency semantics of MPI RMA in an axiomatic model with the goal of formal reasoning on MPI RMA programs. The model defines *actions* in a parallel program related to the corresponding MPI routines. Hoefler et al. [38] define a  $\xrightarrow{co}$  and a  $\xrightarrow{hb}$  order that captures consistency and synchronization of memory accesses. Based on axioms, a valid execution of an MPI RMA program is specified. For a race-free execution, conflicting memory actions *a* and *b* must be ordered by  $\xrightarrow{hb}$  and  $\xrightarrow{co}$ .

The model proposed by Hoefler et al. [38] is similar to the consistency model presented in this thesis but also different in many aspects. First, it focuses specifically on a subset of MPI RMA, as it omits PSCW and request-based RMA operations. The thesis model, instead,

---

<sup>3</sup><https://openucx.org/documentation>

<sup>4</sup><https://www.open-mpi.org>

<sup>5</sup><https://github.com/Sandia-OpenSHMEM/SOS>

<sup>6</sup><https://ofiwg.github.io/libfabric>

addresses RMA models in general, also supporting the notify-wait synchronization of SHMEM and GASPI as well as ordering events such as fences. On the other hand, the MPI RMA model supports both the unified and separate memory models, while the thesis model is focused on the unified memory model only.

The consistency order  $a \xrightarrow{co} b$  in the MPI RMA model is defined as a guarantee that the memory effect of  $a$  will be visible to  $b$  if  $b$  happens arbitrarily late or is repeated. This especially means that  $a \xrightarrow{co} b$  does *not* imply  $a \xrightarrow{hb} b$ , contrary to the assumption of the thesis model where  $\xrightarrow{co}$  is a subset of  $\xrightarrow{hb}$ . The authors state that this  $\xrightarrow{co}$  definition is required to correctly model polling in MPI RMA. In the thesis model, polling in MPI RMA is considered a (benign) data race, as also discussed in Section 2.3.6. Another main difference is the introduction of *virtual actions*. For each remote action issued at the origin (such as remote memory accesses or remote flushes), a *virtual action* is generated that represents the effect of a remote action at the target. In the thesis model, no such virtual actions are required as the modeling as (remote) concurrent regions fully captures the time window when an RMA operation is active.

Although inspired by the axiomatic MPI RMA model, the thesis model uses another notion of  $\xrightarrow{co}$ , modeled as a subset of  $\xrightarrow{hb}$ , as it fits better to the race detection requirements, and it does not use virtual actions, as they are not required in the data race modeling.

#### 4.5.2 coreRMA Language

The coreRMA language [20] provides axiomatic semantics of RMA by modeling the main characteristics of RMA APIs such as UPC [31], MPI RMA, DMAPP [100], and Portals [6]. The language consists of seven basic statements: local read, local write, remote get, remote put, remote get accumulate, compare and swap, and flush. The main idea is to define for those statements corresponding actions, e.g., a remote get involves (1) an external read and (2) a local write. Between those actions, a  $\xrightarrow{hb}$  order where  $a_1 \xrightarrow{hb} a_2$  means that the (memory) effects of  $a_1$  are guaranteed to be visible to  $a_2$ . Additionally, the model specifies the reads-from order  $\xrightarrow{rf}$  where  $w \xrightarrow{rf} r$  means that a read action  $r$  reads the value of a write action  $w$ . Out of a set of axiomatic rules and the  $\xrightarrow{rf}$  order, the  $\xrightarrow{hb}$  order is derived. The axiomatic semantics are used to generate litmus tests in concrete RMA APIs out of possible satisfying program executions in the model to check whether the expected output of the model matches the real output of the litmus tests.

The coreRMA language and the thesis model both capture the essential characteristics of RMA APIs, but with different goals in mind. The coreRMA language is mainly designed to iterate through different possible program execution behaviors in terms of relaxed memory consistency for stress-testing RMA API implementations. Some concepts in coreRMA are too far abstracted for RMA race detection. For example, the only statement in coreRMA for completion is *flush*; it does not distinguish between local and remote completion. A fence construct for ordering is also not included. Further, coreRMA does

not consider the effect of process synchronization and its implied consistency. All those constructs are part of the thesis model and are required to reason on races in RMA.

### 4.5.3 Graph-Based Modeling of RMA Programs

Krzikalla et al. [49–51] model the communication and synchronization semantics of RMA programs with task graphs. The model focuses on GASPI programs but also provides the semantics for MPI RMA and SHMEM routines. Each action in a program is represented as a task that is connected to other tasks in a DAG to represent the  $\xrightarrow{hb}$  relation of the execution. RMA operations create additional *virtual tasks*, e.g., *gaspi\_read* creates a remote read task and a local buffer write task. Local completion operations such as *gaspi\_wait* also create a task that establishes a happened-before edge to the local buffer access. For remote completion, only notify-wait synchronization via *gaspi\_notify* and *gaspi\_notify\_waitsome* is modeled as it is the only way of synchronizing accesses in GASPI. Remote completion mechanisms triggered from the origin side, e.g., *MPI\_Win\_flush*, *MPI\_Win\_unlock* or *shmem\_quiet* are mentioned but not considered in the model. Also, active target completion using *MPI\_Win\_fence* or PSCW is not considered in the model.

The task graph is used for the visualization of memory access patterns in GASPI programs, but also for the detection of non-deterministic executions and for the detection of data races in GASPI programs, as further detailed in [49]. The race detection relies on traversing the task graph post-mortem to find memory accesses that overlap (1) in the address space and (2) in time. While the overlapping in the address space is trivial to compute, the overlap check in time requires finding the earliest start and the latest end of a (remote) memory access, i.e., its concurrent region. For that, the task graph can be traversed along the happened-before edges forward and backward. During the traversal, timestamps (single integer values) are assigned to the tasks and propagated through the graph. Based on the timestamps, the concurrent regions of RMA operations can be determined. Accesses that overlap in time are then checked for conflicts.

Both the task graph model and the model presented in this thesis target a generalized representation of RMA semantics. However, the models differ significantly: First, the task graph model only considers a subset of the MPI RMA and SHMEM functionality. It neglects any completion that is modeled as *collective flush* or *remote flush* in the thesis model. This limits its applicability in the case of MPI RMA and SHMEM programs. Second, the task graph model is primarily intended to visualize PGAS memory access in time-address diagrams to detect performance issues; data race detection is not its primary purpose. Maintaining and recording a task graph is computationally more complex than maintaining vector clocks and the consistency state of RMA operations. Further, as the whole task graph has to be analyzed to find racing accesses, the designed data race detection algorithm only works as a post-mortem analysis. By contrast, the model in this thesis is designed specifically for on-the-fly data race detection and records only the minimum required amount of information. In particular, the vector clocks implicitly encode all required synchronization information without requiring a full task graph.

#### 4.5.4 State-Space Automata for PGAS Semantics

Calin et al. [13] model the execution of PGAS programs as state-space automata. The automaton states represent the current control and memory state of all processes during the program execution. Events in the PGAS program, e.g., load, store, and barrier, are encoded as transitions of the automaton. Based on the defined semantics, the authors define *robustness* as a requirement for a correct PGAS program. It is a finer criterion than the check for data-race freedom, because in some PGAS models such as MPI RMA and UPC, data races are “benign” and do not necessarily lead to an incorrect program execution. They define a happened-before relation (different from the relation defined in this thesis) and prove that in their semantics, a PGAS program is robust if and only if there is no cycle in the happened-before relation. Finally, they provide an algorithm for checking the robustness of a PGAS program and prove that this problem is PSPACE-complete.

The proposed model is mainly used to describe and characterize the robustness problem in PGAS programs. Therefore, it is incomparable to the clock-based race detection model presented in this thesis, which aims for scalable data race detection.

#### 4.5.5 SPMD IR

The SPMD IR [12] is a compiler abstraction layer that encodes the semantics of different SPMD programming models using the multi-level intermediate representation (MLIR) of LLVM [56]. Compiler passes translate the concrete routines of SPMD programming models to common abstract operations in an intermediate representation (IR). Currently, the SPMD IR supports the translation of MPI, SHMEM, and NCCL [71] programs and is designed to be extensible to further programming models.

The outlined use case of such a generalized representation of SPMD semantics is the design of reusable correctness checks based on static analysis for different programming models. For example, the SPMD IR has been used to encode the semantics of collective routines using abstract operations and attributes to specify whether a collective operation is executed by all processes or just a single one. Subsequently, this information is used in a static analysis for the verification of collectives such as missing or mismatched calls for the aforementioned programming models. In an ongoing work [35], the SPMD IR is currently also extended to the detection of local buffer races in MPI RMA and SHMEM. This includes an initial abstraction layer of common RMA semantics.

The SPMD IR is tailored towards static analysis passes in the compiler and targets a broader application field that goes beyond correctness checking. Local buffer race detection in RMA programs is one of its use cases. However, for RMA remote data races, the detection with a static analysis is difficult because the detection requires capturing synchronization and consistency at runtime, which is not possible with a static compiler pass. The clock-based race detection model in this thesis focuses on dynamic runtime

correctness analysis and in particular generic race detection for both local buffer races and remote races in RMA models. Still, the identified primitives presented in the model of this thesis could be used as a blueprint for a similar RMA abstraction layer developed in the SPMD IR.

### 4.5.6 Concurrency Intermediate Verification Language

The Concurrency Intermediate Verification Language [97] provides an extended C language, named CIVL-C, enriched with constructs to represent concurrency properties. It captures the concurrency of different parallel programming models such as message-passing via MPI, multithreading via OpenMP and pthreads, and GPU parallelism via CUDA. CIVL-C is used as a framework for formal verification of correctness properties such as deadlock-freedom and functional equivalence checks of parallel programs.

The concrete primitives of the programming models are first translated to corresponding pure CIVL-C primitives. Then, the resulting CIVL-C program is translated into a graph-based intermediate representation capturing different semantic properties of the program in states, e.g., consisting of *scopes* containing relevant variables, and the set of processes with their call stack. This representation is subsequently used to perform model checking and symbolic execution analyzing the different possible state transitions, i.e., the different execution interleavings, to check for errors such as deadlocks, data races, and memory leaks in the program. Further, CIVL can check for functional equivalence between two programs, i.e., whether two programs produce the same result for a given input set. Thus, it can verify if a parallelized program is functionally equivalent to its sequential counterpart. By abstracting the parallel programming models to a generic abstraction language, CIVL also supports the verification of hybrid programs such as MPI+OpenMP.

The concurrency modeling approach of CIVL is similar to the RMA modeling approach in the sense that it tries to find common semantic properties in different parallel programming models. However, CIVL is different from the RMA consistency and race detection model presented in this thesis in many ways: First, CIVL-C focuses on representing *any* kind of concurrency (message passing, threading, GPU parallelism) in a single language. The thesis model focuses on modeling the semantics of RMA programming models with the goal of data race detection in RMA programs. Further, CIVL-C only covers the basic features of the programming models: For MPI, only blocking point-to-point operations and collectives are supported, but no non-blocking operations. Especially, RMA is currently not supported which also means that CIVL is not capable to detect races in RMA. In general, the CIVL approach is tailored towards symbolic execution and model checking which has different challenges such as state space explosion that need to be addressed. The RMA race detection model is focused on being used in an on-the-fly data race analysis for a specific program run instead of going over all possible execution paths of a program.

### 4.5.7 Race Detection Models

The race detection model presented in this chapter relies on vector clocks to determine the concurrent regions of accesses and verify whether conflicting accesses are concurrent. There are several other race detectors that implicitly use a similar model of detecting concurrent regions. Those approaches will be briefly summarized in the following. A detailed technical description and comparison of the different race detectors will be presented in Chapter 5.

MC-Checker [16] is a post-mortem MPI RMA race detector. It records the program execution at runtime and based on the trace, it converts the events (memory accesses, completion calls, synchronization calls) to a set of vertices that are interconnected to a directed acyclic graph (DAG) according to the happened-before relation. The race detection is then a graph traversal on the DAG. To reduce the overhead, MC-Checker ignores transitive synchronization, e.g., a chain of send and receive operations between different processes, collectives (except barriers), and non-blocking operations, which may lead to false positives. MC-CChecker [22] is an independent modification of MC-Checker that uses encoded vector clocks [52] instead of a DAG to find conflicting accesses. It timestamps events in a post-mortem analysis and does a pair-wise comparison of concurrent regions to check for conflicts. Both MC-Checker and MC-CChecker are designed for a post-mortem analysis with global knowledge about all accesses, while the race detection model in this thesis is designed for on-the-fly detection.

PARCOACH [2, 81] is an MPI correctness framework that provides an on-the-fly race detector for MPI RMA. The RMA race detection only works on programs with bulk-synchronous completion semantics, i.e., collective completion in terms of the thesis model, e.g., *MPI\_Win\_fence*. Due to this assumption, the race detection just has to collect all memory accesses starting from one collective completion up to the next collective completion and check for conflicting memory accesses. After collective completion, all previous memory accesses do not have to be considered any further as they cannot race with later accesses. While this race detection scheme is simple, it only works on a very limited set of RMA programs. A similar assumption is used by Park and Chung [74] which is another on-the-fly MPI RMA race detector that can detect races for *MPI\_Win\_fence* and PSCW. Both approaches differ from the race detection defined in this thesis, as their simplifying assumptions do not require recording individual completion and synchronization states between processes.

UPC-Thrille [73] is an active testing approach for races in UPC programs. It uses a combination of a lockset-based and a happened-before analysis that checks for potential races in the first execution of the program and verifies the potential race by enforcing it in a second execution. The approach differs from all previously explained race detection approaches since it executes the program several times to check for potential races.

## 4.6 Results and Discussion

This chapter provides a formal model to reason on the consistency of RMA operations in RMA programming models. The model extends the well-known happened-before order  $\xrightarrow{hb}$  to the consistency order  $\xrightarrow{co}$  to incorporate the completion semantics of RMA models. It defines seven generic completion events *collective flush*, *local flush*, *remote flush*, *fence*, *target notify*, *target wait*, and *local complete* to represent the different completion modes present in the RMA models MPI RMA, OpenSHMEM, and GASPI. The effect of the completion events on the  $\xrightarrow{co}$  order is formalized through inference rules. As part of the model, a mapping of the concrete completion routines of MPI RMA, OpenSHMEM, and GASPI to the generic completion events is provided. It proves that the consistency semantics of those three models can be abstracted without losing substantial information for subsequent data race detection. Also, an outlook to other RMA approaches, e.g., UPC++ and Coarray Fortran, has shown that their consistency semantics could also be mapped to the generic completion events of the model.

The consistency model is combined with the clock-based synchronization analysis to define an RMA race detection model. For any RMA operation, the *concurrent region* defines the earliest and the latest point in the execution at which an RMA operation's effect may occur. The concurrent region of an RMA operation can be determined on the fly by observing completion and synchronization events. To find the corresponding start and end points of a concurrent region, vector clocks are used. The defined race detection algorithm finally checks for overlapping concurrent regions of conflicting accesses to find RMA races.

The race detection model omits some model-specific details, such as distinguishing MPI RMA windows or SHMEM context. Although those concepts could be incorporated in the model, it would require a significant additional notation effort that is avoided to keep the model simple. The implemented race detector still considers those details that were omitted in the model.

The presented race detection model is the first one that uses vector clocks to find data races in RMA programs on the fly. It supports the largest number of synchronization and completion modes among all previous race detection models, which typically focus on a smaller subset of a single RMA programming model. The versatility of the model is proven by implementing it in an RMA race detection tool that will be presented in the next chapter.

## 5 Scalable Race Detection for RMA Programs

The race detection model presented in Chapter 4 describes how to detect data races in RMA programs using concurrent regions. In practice, implementing this model in a scalable tool workflow to work on the fly on RMA applications proves challenging. RMA applications are typically designed to scale up to thousands of processes, so a scalable implementation has to distribute the race analysis across the executing processes. Further, each RMA operation is issued on the origin and modifies a memory location at the target, where it may interfere with (1) local memory accesses of the target itself or (2) RMA operations from other processes. Therefore, performing the actual race detection between all memory accesses *at the target* is a natural choice. The one-sided nature of RMA operations requires the origin to transmit metadata about ongoing remote accesses and their consistency state on the fly to the target. Simultaneously, synchronization between processes has to be captured to understand the remote concurrent regions of RMA operations at the target. All of this information is tracked and combined in *RMASanitizer*, an on-the-fly data race detector for RMA programs, presented in this chapter. The tool is open-source and publicly available under a BSD 3-clause license, details are provided in Appendix A.

*RMASanitizer* performs data race detection on RMA programs at runtime and supports MPI RMA, SHMEM, and GASPI. It uses the vector clock exchange mechanism introduced in Chapter 3 to track synchronization between processes. To track the consistency state of RMA operations, it intercepts all relevant RMA routines and computes the concurrent regions as described in the race detection model in Chapter 4. To exchange metadata about ongoing RMA operations between processes, *RMASanitizer* utilizes the MUST [36] and GTI [37] infrastructure. For race detection, it annotates the computed concurrent regions to the shared-memory race detector ThreadSanitizer [95]. To reduce the analysis overhead, *RMASanitizer* avoids the instrumentation of irrelevant local memory accesses by leveraging a static analysis on the code.

This chapter is structured as follows: Section 5.1 gives an overview of existing work on RMA race detection. Section 5.2 introduces the shared-memory race detector ThreadSanitizer, which is an essential component of *RMASanitizer*. The implementation and architecture of *RMASanitizer* are explained in Section 5.3, including the used instrumentation techniques, the detection and annotation of concurrent regions, and the implemented race detection workflow. The contents of this section are based on my previous publications [89, 93]. Finally, Section 5.4 summarizes the results and discusses future work.

## 5.1 Related Work on RMA Race Detection

This section gives an overview of existing work on RMA race detection discussing different analysis approaches and their limitations. Most related work has focused on MPI RMA race detection, while fewer approaches are dedicated to SHMEM and GASPI.

### 5.1.1 Post-Mortem DAG Traversal

MC-Checker [16] is a post-mortem race detector for MPI RMA programs. It supports all RMA constructs of MPI 2.2, i.e., *MPI\_Win\_fence*, PSCW, and *MPI\_Win\_lock/unlock*. The race detection runs in two phases: First, MC-Checker traces all relevant events at runtime, such as local memory accesses, MPI RMA operations, MPI RMA completion calls, and MPI synchronization operations. Second, the resulting trace is analyzed post-mortem to construct a directed acyclic graph (DAG), where vertices represent the recorded runtime events and edges represent the happened-before order. For the DAG construction, MC-Checker simulates (based on the trace) the progress of the MPI processes, thereby analyzing matching synchronization, e.g., via barriers or send/receive pairs. Based on the DAG and the consistency information of the RMA accesses, MC-Checker identifies subgraphs in the DAG that may execute concurrently, also called *concurrent regions*. For race detection, it checks the concurrent regions for conflicting accesses.

MC-Checker instruments memory accesses and MPI calls via Clang compiler instrumentation. To reduce the slowdown of the profiling step, a static analysis is employed to instrument only *relevant* local memory accesses. The analysis examines the source code to mark those variables as “relevant” that are passed to an MPI RMA window allocation call or used as a local buffer in an RMA call. Those markers are then propagated to other variables by following pointer assignments or function calls. Finally, all marked variables are then instrumented. The analysis is implemented in the Clang frontend and is, therefore, limited to C/C++ programs.

MC-Checker successfully detects races in simple test programs and races injected into RMA applications. The profiling slowdown on real-world applications ranges from 1.5x to 3.0x for runs with up to 128 processes [16]. The authors have not reported the required time for the post-mortem analysis or the trace size. However, as with every post-mortem race detector, the traces of the profiling run may become unmanageable large for long-running applications.

MC-Checker is limited in the detection of synchronization in MPI: For efficiency reasons, the matching computation to construct the DAG does not consider transitive synchronization effects, e.g., a chain of send-receive pairs. Further, not all collective and non-blocking operations are considered potential synchronization sources. Ignoring synchronization effects may lead to falsely reported races (false positives). Also, as MC-Checker relies on the MPI 2.2 standard, some synchronization calls introduced with MPI 3.0, such as *MPI\_Win\_flush*, are not considered but could be incorporated into the model.

### 5.1.2 Post-Mortem Timestamping

MC-Checker does not consider transitive synchronization in MPI processes, which may lead to false positives in the race detection. MC-CChecker [22] is an independent extension of MC-Checker that overcomes this limitation by using encoded vector clocks [52] instead of a DAG to find concurrent regions. The profiling step is identical to MC-Checker, while the post-mortem analysis step uses a timestamping system for MPI RMA based on vector clocks. It captures the synchronization through MPI barriers, fences, send/receive pairs, and PSCW. MC-CChecker extracts the events from the trace files and uses the timestamping system to check for concurrent accesses. As the authors show, this eliminates false positives in transitive synchronization scenarios. MC-CChecker still does not consider the (potential) synchronization effect of non-blocking operations and collectives other than *MPI\_Barrier* and *MPI\_Win\_fence*. Further, the integration of MPI 3.0 features such as *MPI\_Win\_flush* is discussed but not implemented.

### 5.1.3 Post-Mortem Task Graphs

A similar approach to MC-Checker is the task graph model presented by Krzikalla et al. [49–51]. It represents events in an RMA program as vertices and the happened-before relation as edges between the tasks. The model has been discussed in detail in Section 4.5.3. It is significantly different from the MC-Checker model, as it focuses on the notify-wait synchronization of GASPI, which MPI RMA does not support, and introduces virtual tasks to represent events happening at the target process. The race detection requires traversing the task graph forward and backward to find concurrent RMA operations and local memory accesses.

The model has been implemented in a prototype race detection tool for GASPI programs [49]. It uses dynamic binary instrumentation via Intel Pin [64] to record memory accesses, and GASPI calls at runtime and saves the events in a trace. The task graph is created out of the traces and then traversed to detect races. During the traversal, timestamps are assigned to the tasks and then used to find concurrent accesses.

The authors evaluated the tool on GASPI proxy apps with up to 192 processes [49]. The recording of runtime events leads to application slowdowns of up to 10x and trace sizes of 20 – 30 GB when the application runs for only 60 seconds. Similarly, the post-mortem graph traversal shows scalability problems as it takes orders of magnitudes longer than the actual program execution, especially for larger process numbers.

### 5.1.4 On-the-Fly Detection for Bulk-Synchronous Programs

PARCOACH [81] is a correctness checking tool for MPI collectives that has been extended to data race checks in MPI RMA programs in a later work [2]. The race detection analysis focuses on MPI RMA programs that use completion and synchronization calls

collectively, i.e., *MPI\_Win\_fence* or collectively called *MPI\_Win\_lock/unlock\_all* combined with an *MPI\_Barrier*. Fine-granular completion, e.g., via individual *MPI\_Win\_lock* or *MPI\_Win\_flush*, or synchronization, e.g., via send-receive pairs, is not supported.

The core idea of the analysis is that each MPI process manages a binary search tree (BST) of memory access intervals. For each RMA window, a process creates a BST storing all local memory accesses and remote accesses to the exposed memory region. Whenever a (relevant) memory access occurs, the BST is first traversed to find overlapping conflicting intervals of previous memory accesses. If no conflict is detected, the access is subsequently added to the BST. Otherwise, the execution is aborted with a race report. While the local memory accesses to the RMA window can be intercepted locally for each process, remote accesses issued at the origin have to be transmitted to the target. An additional thread is created for each MPI process that continuously polls with *MPI\_Recv* for remote access information transmitted by the origin via *MPI\_Send*. Whenever an MPI RMA epoch ends, each process collects the total number of memory accesses it should have received from other threads (via *MPI\_Reduce*) and stalls the execution until all (outstanding) messages have been received. Then, the BST is completely cleared and the next MPI RMA epoch in the application can start.

The implementation relies on the MPI profiling interface PMPI to intercept the MPI RMA calls. Local memory accesses are instrumented with an LLVM compiler pass that works for C/C++ and Fortran programs. To avoid instrumenting all local memory accesses, the pass checks for each defined function in the source code if it invokes an MPI RMA call that opens an RMA epoch. If this is the case, then it instruments all memory accesses occurring after the call in that function.

A significant overhead source is the size of the BST and the number of instrumented local memory accesses. A follow-up publication [106] improves the insertion algorithm for the BST by merging adjacent memory intervals to reduce its size. Further, it avoids some false positives and negatives by adjusting the overlap check algorithm. Another follow-up publication [107] further tries to reduce the overhead by compressing regular memory access patterns in the BST. Moreover, using an alias analysis, the local memory access instrumentation is optimized to instrument only load and store instructions to the RMA window memory. This may lead to missed local buffer races, but the authors argue that a local buffer race could already be detected by their static race analysis approach proposed in another publication [82].

The main limitation of PARCOACH is that it only works for MPI RMA programs with collective synchronization. This simplifies the race detection because after each collective synchronization, all RMA operations are assumed to be completed, so the BST can be cleared. Any fine-grained synchronization via MPI window locks, PSCW, or a notification-based mechanism is not supported. In [107], a first step was made to also consider fine-grained completion with *MPI\_Win\_flush* in the BST analysis, but it is still limited to specific use cases. Further, the filtered memory access instrumentation used in PARCOACH does not consider interprocedural dependencies. Functions accessing relevant memory addresses but not calling any MPI RMA routine are ignored, leading to many missed races

(false negatives). Section 6.1.3 discusses details of this limitation. Moreover, a detailed comparison of PARCOACH to RMA Sanitizer is provided in Chapter 6.

### 5.1.5 On-the-Fly Mirror Windows

Park and Chung [74] propose a method based on mirror windows to detect data races in MPI RMA programs. When an MPI RMA window is allocated, a corresponding mirror window is created that tracks the status of the window memory. For each entry in the original RMA window, an entry in the mirror window can have the state “no-op”, “read”, or “write”. RMA operations issued at the origin correspondingly update the state of the window memory at the target. The tool uses the MPI profiling interface PMPI to intercept the MPI RMA calls. On every RMA operation such as *MPI\_Put*, the origin first gets the current state of the addressed location at the target mirror window using *PMPI\_Get*, checks for a data race, and then updates the mirror window using *PMPI\_Put*.

The proposed race detection can only detect data races between remote accesses, as local memory accesses are not considered at all. Further, the algorithm only works with active target synchronization, i.e., *MPI\_Win\_fence* and PSCW, as it relies on resetting the mirror window state to “no-op” whenever the exposure epoch of a window ends. Thus, passive target synchronization, where the target is unaware of the remote completion, is not supported. Lastly, the presented race detection internally uses concurrent *MPI\_Get* and *MPI\_Put* accesses to the mirror window entries, which themselves lead to data races in the tool.

### 5.1.6 Active Testing

UPC-Thrille [73] is an active testing approach to find data races in UPC [31] programs. The race detection is performed in two steps: In the first step, the race prediction phase, an imprecise dynamic race analysis combines a lockset and a barrier counter to find potentially concurrent conflicting events. The resulting set contains tuples of two memory accesses that may lead to a race. For each memory access, metadata describing the accessed memory address, the accessing task, the set of held locks, the access type (read/write), and the program statement is stored. In the second step, the race confirmation phase, the program is re-executed for each tuple of potential races. The program schedule is influenced by the insertion of delays to enforce and confirm the data race.

To record the different memory accesses and synchronization events, the source code is instrumented with calls into the UPC-Thrille runtime. An evaluation on the NPB benchmarks ported to UPC and executed with up to 1024 processes shows that the slowdown of the race prediction phase is at maximum 1.2x.

A problem with this approach is the number of detected false positives verified in the race confirmation phase: For the NPB benchmarks, the race prediction phase detected

several potential races in race-free applications. In the race confirmation phase, those races were then correctly classified as false positives. However, for each potential race that later turns out to be a false positive, the application has to be re-executed. Thus, applying the tool to long program runs with many false positives is infeasible, as the race confirmation phase will take a significant amount of time.

### 5.1.7 Static Analysis

There have been several approaches to detect RMA data races using static analyses of the source code. Saillard et al. [82] perform a breadth-first search on the CFG of the program to find local buffer races in MPI RMA programs. For that, the algorithm maintains a list of observed non-completed MPI RMA calls during the traversal for each target process. On every local memory access or RMA operation, the current access is compared with previous non-completed RMA operations to find conflicts. On a completion call such as *MPI\_Win\_flush*, the list is truncated. The analysis is implemented as LLVM pass running on the LLVM Intermediate Representation (IR) and works on C/C++ and Fortran programs. Remote memory accesses and, therefore, remote races are not considered since mapping remote accesses to the concrete locations in the target window with static analysis is challenging. As with every static analysis, any control flow or parameter that depends on runtime information may lead to false negatives or false positives in the detection.

OpenSHMEM-Checker [5] is a static analysis tool for OpenSHMEM programs. It is based on the Clang Static Analyzer framework that can perform path-sensitive analyses. It provides violation checks, such as passing a private memory address where a symmetric memory address, i.e., a memory address in the remotely accessible memory space, is expected. Further, it also has experimental support for checking for data races in SHMEM programs. The race detection requires that the accessed memory region of a SHMEM communication call can be determined at compile time (via constant propagation). The accessed region is then symbolically represented and marked as “unsynchronized”. If another conflicting access occurs before a SHMEM synchronization call is invoked, a data race is detected. The authors evaluated their check on a matrix multiplication code where they injected a data race. The check could not find the data race, because the accessed locations and the targeted PE numbers could not be propagated to constants. An extension to support non-constant accesses is postponed to future work.

MPIRace [109] is another static data race detector for MPI programs working on the LLVM Intermediate Representation (IR). It focuses on data races with non-blocking point-to-point communication, e.g., modifying the communication buffer of an *MPI\_Isend* before the operation is completed. Still, the authors also mention the application of their approach to MPI RMA programs. MPIRace first identifies so-called May-Have-Races (MHR) regions: They represent code regions in which an MPI routine might perform memory accesses concurrently to the program itself. For example, an *MPI\_Isend* starts an MHR region and a matching *MPI\_Wait* ends it. Together with an analysis of the memory accesses in the MHR regions, MPIRace can detect potential data races. The analysis scans

through the LLVM IR representation to find the MHR regions and memory accesses. For MPI RMA accesses, the detection of MHR regions and their analysis are briefly mentioned by the authors, but not further evaluated. Due to MPIRace being a static analysis, missing runtime information may lead to false negatives or false positives in the detection.

### 5.1.8 Model Checking Approaches

Different model-checking approaches for PGAS programs have been proposed, such as UPC-SPIN [23] and XMP-SPIN [1] that leverage the model checker SPIN [39] to verify correctness conditions such as race-freedom in UPC [31] and XMP [58], respectively. Both approaches require the programmer to manually specify the conditions to be checked along with the source code which SPIN then verifies. The authors could prove race-freedom on small UPC and XMP kernels when checking the execution with a few processes. Due to the state space explosion problem of model checking, the method can only be applied to smaller subparts of a real-world program. Further, the manual specification of the conditions to be checked is a high effort for the programmer.

### 5.1.9 Nasty-MPI

Nasty-MPI [48] is a debugging tool that tries to enforce synchronization errors in MPI RMA programs. It intercepts all MPI RMA calls using the MPI profiling interface PMPI and explicitly delays forwarding them to the MPI library while still adhering to the MPI RMA semantics. For example, an *MPI\_Get* call is intercepted and only forwarded to the MPI library as late as possible, i.e., when a corresponding RMA completion call such as *MPI\_Win\_flush* is issued by the program. Further, RMA operations in the same epoch are reordered and RMA operations with a larger count of accessed elements are split into smaller ones. All of those modifications try to uncover synchronization errors in the program that might otherwise only become visible in certain MPI implementations or on certain systems. Nasty-MPI is not a race detection tool, but it may be combined with the presented dynamic race detectors to increase the chance of detecting a data race.

### 5.1.10 Discussion

The presented related race detection approaches mainly focus on MPI RMA race detection, and most of them only support a subset of synchronization and consistency primitives. The on-the-fly RMA race detector implemented in the context of this thesis, RMASanitizer, covers all major synchronization and consistency primitives and implements the abstracted race detection model presented in Chapter 4 to support MPI RMA, SHMEM, and GASPI. In terms of generalized race detection, only the task-based approach of Krzikalla et al. [50] follows a similar goal as this thesis, but focuses on post-mortem race detection and the implementation itself only supports GASPI. MC-Checker also uses a

notion of concurrent regions for MPI RMA operations similar to RMA Sanitizer, but also only in the context of post-mortem race detection which has different requirements as the concurrent regions can be constructed with global knowledge based on the trace.

The only other race detector providing an on-the-fly RMA race analysis is PARCOACH. However, it is limited to MPI RMA and only supports programs with collective synchronization. A detailed comparison between PARCOACH and RMA Sanitizer is provided in Chapter 6.

## 5.2 Shared-Memory Race Detection with ThreadSanitizer

The implementation of RMA Sanitizer, which will be presented in Section 5.3, partly relies on ThreadSanitizer for the RMA race detection, so this section briefly introduces its architecture. ThreadSanitizer (TSan) [94, 95] is an on-the-fly shared memory race detector that uses shadow memory to efficiently find data races in multithreaded programs with POSIX threads at runtime. It intercepts all memory accesses and relevant thread synchronization calls to track the synchronization state of accesses. In its first version [94], it uses a hybrid happened-before and lockset-based analysis and relies on dynamic binary instrumentation with Valgrind [66]. In its second version [95], it solely relies on happened-before analysis, and the dynamic binary instrumentation was replaced by a compile-time instrumentation using the GNU or LLVM compiler, which significantly reduces the slowdown. The third version<sup>1</sup> further reduces the memory consumption of the shadow memory and vectorizes the race detection algorithm. In the following, the properties of the third version integrated in the LLVM toolchain will be described as this is the version that RMA Sanitizer relies on.

### 5.2.1 Instrumentation

Prior to the runtime analysis of the program, the memory accesses in the source code are instrumented in a compile-time instrumentation step. For that, TSan prepends calls to the runtime library such as `__tsan_write4(addr)` to every memory access, as shown in Figure 5.1. The instrumentation itself is done on the LLVM Intermediate Representation (LLVM IR) level. Further, it inserts calls to track function enter and exit events to generate stack traces when a race is detected. TSan avoids instrumenting redundant memory accesses, e.g., a read before a write to the same memory address. It also ignores memory accesses that can be proven not to lead to a race, e.g., stack variables that can be proven not to be accessed by other threads.

---

<sup>1</sup><https://reviews.llvm.org/D112603>

```

1 void *thread1(void *x) {
2   __tsan_func_entry(returnaddress);
3   __tsan_write4(&myglobal);
4   myglobal = 42;
5   __tsan_func_exit();
6
7   return x;
8 }

```

(a) Instrumented source code (pseudocode).

```

1 define ptr @thread1(ptr %0) #0 {
2   %2 = call ptr @llvm.returnaddress(i32 0)
3   call void @__tsan_func_entry(ptr %2)
4   call void @__tsan_write4(ptr @myglobal)
5   store i32 42, ptr @myglobal, align 4
6   call void @__tsan_func_exit()
7   ret ptr %0
8 }

```

(b) Instrumented LLVM IR.

Figure 5.1: Source code instrumentation with TSan that inserts calls to the TSan runtime library for memory access and stack tracing. The right variant shows how the instrumentation is actually done on LLVM IR, while the left variant is an equivalent C source code for illustration purposes.

In addition to memory access instrumentation, TSan tracks relevant synchronization calls that ensure happened-before ordering between threads, e.g., pthread synchronization and C++ thread synchronization. For that, it wraps the relevant functions at runtime and considers their synchronization effect (signal, wait) for the race detection.

## 5.2.2 State Machine

The actual race detection in TSan implements a state machine [101] that reacts to all relevant events, i.e., memory accesses and synchronization events, during the program execution and performs a happened-before analysis to find data races. To capture the happened-before order between threads, TSan stores a vector clock for each thread that is updated on memory accesses and synchronization events. To store the metadata on accessed memory locations, TSan relies on shadow memory where each 8-byte application memory is mapped to a *shadow state*, which in turn consists of  $N$  shadow words. Each *shadow word* corresponds to some memory access in the 8-byte application memory and consists of the components shown in Figure 5.2. Since each shadow state carries  $N$  shadow words, where  $N$  is by default set to 4, each state can store a maximum of 4 different memory accesses. On each memory access, the state machine performs the following actions [101]:

1. Increment the local vector clock entry of the thread that executes the memory access.
2. Create a new shadow word for the given memory access.
3. Map the accessed memory address to the corresponding shadow state and check whether the new shadow word races with any stored shadow word already present in the shadow state. There is a race between two memory accesses if the following conditions are all true:
  - a) The accessed memory access regions (within the 8-byte application memory) are overlapping.

## 5 Scalable Race Detection for RMA Programs

- b) At least one access is a write.
  - c) The accesses originate from different threads.
  - d) The accesses are not happened-before ordered, i.e., concurrent.
4. Store the new shadow word in an empty slot in the shadow state or, if all slots are occupied, replace a random old shadow word with the new one.

The previously described shadow word comparison uses vector instructions to compare multiple shadow words using a set of SIMD instructions. For a scalable analysis of the happened-before order, the concurrency of accesses is determined using FastTrack [26], also referred to as “fast happened-before”. Instead of storing a full vector clock with each memory access in a shadow word, TSan only stores the thread ID of the accessing thread and the current local (scalar) clock value of the thread, named “epoch”. The concurrency analysis of two accesses can then be done in a constant-time operation by comparing the epoch and thread ID of the stored memory access with the synchronization state of the thread performing the new memory access.

False positives may occur in TSan if synchronization between threads is missed. This can happen if synchronization is done in non-instrumented code, e.g., external libraries, that TSan cannot observe. Further, as a result of using the FastTrack algorithm, TSan stores different memory accesses to the same memory location, (potentially) from different threads with different synchronization states, in different shadow words. False negatives may occur due to the limited storage capacity of four shadow words per shadow state. If a racing shadow word is evicted from the shadow state before another racing access occurs, then the race might be missed. This is in particular true for concurrent reads from different threads to the same memory location: If a read from one thread is concurrent to a later write by another thread, it might be that the read is evicted by reads from other threads before the conflicting write is issued, missing the race. To avoid those false negatives, the history size, i.e., the size of the shadow state, can be increased from 4 to a larger value.

As with every happened-before race detector, TSan only records the synchronization state of a particular execution, so in case of non-deterministic synchronization, races that only occur in another synchronization interleaving would be missed.

|                            |                 |                 |                 |                   |
|----------------------------|-----------------|-----------------|-----------------|-------------------|
| Access (Size/Pos)<br>8 bit | SlotID<br>8 bit | Epoch<br>14 bit | isRead<br>1 bit | isAtomic<br>1 bit |
|----------------------------|-----------------|-----------------|-----------------|-------------------|

Figure 5.2: Shadow word (32 bit) storing metadata of a memory access in ThreadSanitizer (version 3) [102]. *Access* stores information about the size and position of the memory access, *SlotID* the thread ID of the access, and *Epoch* a scalar clock for the FastTrack happened-before analysis.

### 5.2.3 User Annotations

As the instrumentation of TSan might not catch all synchronization mechanisms and memory accesses, users can interface with TSan's annotation API [94] to annotate them. The annotation API provides functions that the user can manually insert into the source code. The API has been introduced with the first TSan version [94] and also works with all newer TSan versions. The following calls are in particular relevant:

- `AnnotateHappensBefore(identifier)`: Annotate a signal from the calling thread using the given identifier.
- `AnnotateHappensAfter(identifier)`: Annotate a wait from the calling thread using the given identifier.
- `__tsan_read_range(addr, size)`: Annotate a read access to a given address with a given size.
- `__tsan_write_range(addr, size)`: Annotate a write access to a given address with a given size.

With *AnnotateHappensBefore*, the TSan runtime stores the current vector clock of the calling thread at the resource with the given identifier, also named *synchronization clock* in the following. A matching *AnnotateHappensAfter* with the same identifier makes the calling thread retrieve the stored synchronization clock and merge it with its current vector clock, effectively establishing a happened-before order between the two annotation calls. This pattern represents resource-based synchronization as discussed in detail in Section 3.4.5.

The annotation API is a flexible mechanism that makes TSan work with further synchronization and threading models besides pthreads and C++ threads. In particular, Archer [3] extends TSan to support threading with OpenMP. The OpenMP Tools Interface [7, §19] delivers detailed runtime information about the synchronization state of OpenMP threads, e.g., barriers, critical sections, and locks. Archer adds this synchronization information to TSan using the aforementioned annotation calls.

### 5.2.4 Fibers

In higher-level programming models, there might be several execution contexts associated with a single thread, also named thread-local concurrency [79]. For example, when MPI non-blocking communication is used, an *MPI\_Isend* initiates a non-blocking send and a matching *MPI\_Wait* completes it. Between initiation and completion, the MPI library might at any time read the local buffer with the data to be sent. This memory access happens asynchronously to the executing thread in the MPI library, i.e., even in a single-threaded execution from the user perspective, the access might be asynchronous. Another example is OpenMP tasking where tasks scheduled to the *same* thread would be interpreted as sequential by TSan, but could also be scheduled to different threads and

## 5 Scalable Race Detection for RMA Programs

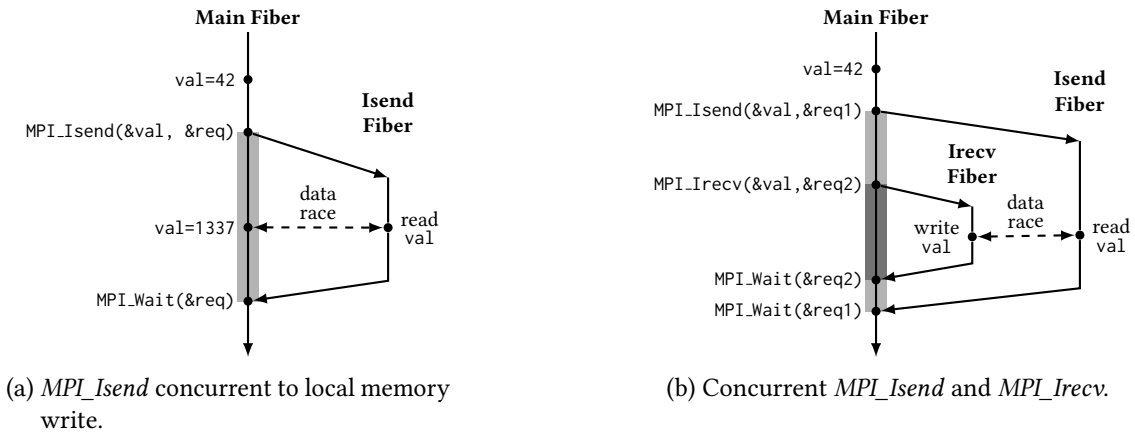


Figure 5.3: Fiber examples for MPI non-blocking point-to-point communication. The concurrent regions of the non-blocking MPI operations are gray-shaded.

therefore executed concurrently. If TSan was aware of the different execution contexts, it could interpret this concurrency correctly.

To incorporate thread-local concurrency, ThreadSanitizer supports the annotation of *fibers*<sup>2</sup>. Each physical thread consists of exactly one fiber by default, the main fiber. Additional fibers can be created at runtime and the execution context can be switched dynamically with the following user annotation API:

- `void* __tsan_get_current_fiber()`: Return the currently active fiber.
- `void __tsan_create_fiber(unsigned flags)`: Create a new fiber.
- `void* __tsan_destroy_fiber(void* fiber)`: Destroy the given fiber.
- `void __tsan_switch_to_fiber(void* fiber, unsigned flags)`: Switch the execution to the given fiber.

With those annotation calls, non-blocking MPI operations can be annotated as independent fibers to ThreadSanitizer, as shown in Figure 5.3. On each non-blocking MPI call, a fiber within the current thread is created and the memory access to the local buffer is annotated to that fiber, as also detailed in [45]. This allows TSan to detect the data race, although the execution itself is single-threaded.

The concurrent region concept for RMA operations introduced in Section 4.2 matches the fiber concept of TSan. In RMA Sanitizer, the concurrent regions of RMA operations are annotated to fibers in TSan. The details of the translation from concurrent regions to fibers will be discussed in Section 5.3.4.

<sup>2</sup><https://reviews.llvm.org/D54889>

## 5.3 RMA Sanitizer Architecture

In the context of my research, I have implemented an on-the-fly race detector named *RMA Sanitizer* [93], which implements the RMA race detection model described in Chapter 4. RMA Sanitizer can detect data races in MPI RMA, SHMEM, and GASPI programs written in C/C++ and Fortran. It is an extended version of my previous work MUST-RMA [88, 90] that also uses the concurrent region approach, but only works with MPI RMA. RMA Sanitizer runs concurrently with the RMA application and can detect RMA races at execution time. It can detect both, local buffer and remote races. Its primary goal is a high race detection accuracy while still being applicable to real-world applications in terms of overhead.

RMA Sanitizer combines a static analysis with a dynamic runtime analysis as illustrated in Figure 5.4. The source code is first translated with an LLVM frontend (Clang or Flang) to the LLVM Intermediate Representation (LLVM IR). A static analysis finds and instruments those load and store instructions relevant to RMA race detection, which avoids unnecessary runtime analysis overhead. The static analysis is a modified variant of the ThreadSanitizer instrumentation described in Section 5.2.1. Subsequently, the runtime analysis checks for data races during the execution of the instrumented application. The runtime analysis combines the vector clock analysis presented in Chapter 3 with the race detection model presented in Chapter 4 to analyze the concurrent regions of RMA operations. The implementation utilizes the correctness tool infrastructure provided by MUST [36] and GTI [37]. Finally, the runtime analysis annotates the concurrent regions of RMA operations as fibers to ThreadSanitizer. Upon a detected race by ThreadSanitizer, RMA Sanitizer validates the race and reports the affected source code lines and processes.

The following subsections iterate over the different components of RMA Sanitizer following the toolchain workflow in Figure 5.4. Section 5.3.1 explains how the static analysis detects and instruments relevant local memory accesses through data-flow analyses

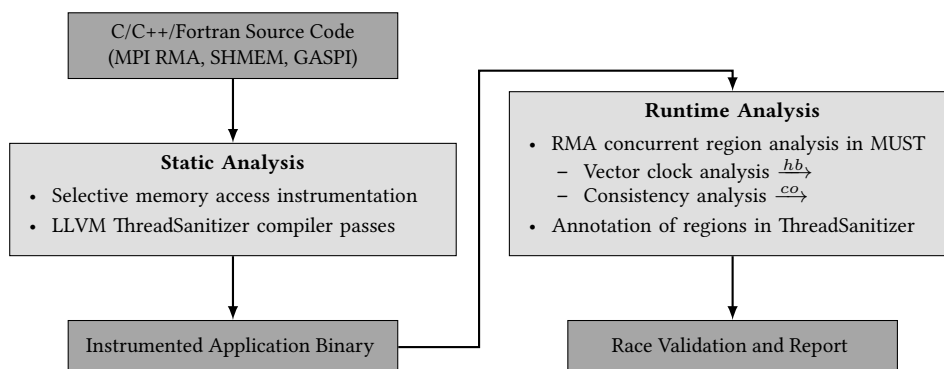


Figure 5.4: RMA Sanitizer toolchain workflow. Relevant local memory accesses are instrumented at compile time with LLVM. The clock-based runtime analysis finally detects the RMA data races.

on the LLVM IR. In Section 5.3.2, the infrastructure of MUST and GTI used to track the synchronization and consistency of RMA operations is explained. Section 5.3.3 and Section 5.3.4 describe how the concurrent regions are detected and annotated as fibers to ThreadSanitizer. Section 5.3.5 summarizes the implemented race detection workflow. Finally, Section 5.3.6 discusses an alternative race detection mechanism using Interval Skip Lists instead of ThreadSanitizer’s shadow memory.

### 5.3.1 Selective Local Memory Access Instrumentation

Since RMA operations may interact with local memory accesses, an RMA race detector also has to track them to get a full picture of the execution. The principle of memory access instrumentation used by RMA Sanitizer is the same as that of ThreadSanitizer described in Section 5.2.1, i.e., the memory accesses in the source code are extended by calls into a runtime library. As previous work [2, 16, 90, 92] has shown, instrumenting *all* local memory accesses leads to a massive overhead in the subsequent runtime analysis. Many local memory accesses are typically irrelevant for RMA race detection. This section shows how static analysis at compile time can be leveraged to identify such irrelevant local memory accesses. The section summarizes the results I have published in [93] as a result of the supervision of Yussur Mustafa Oraji’s bachelor thesis [72].

Figure 5.5 shows a motivating example for the static filtering of irrelevant memory accesses in RMA race detection. It is a 1D-stencil code, a simplified variant of a 2D-stencil defined in [105], using MPI RMA for the halo exchange. The arrays  $U$  and  $U_{new}$  are used for the stencil computation, and the arrays  $bufOut$  and  $bufIn$  for the halo exchange together with an RMA window to exchange data with the left and right neighbor. In each iteration, each process first copies its halo data to  $bufOut$  (lines 26 – 27) and then performs the halo exchange with the left and right neighbor using fences and  $MPI\_Put$  (lines 6 – 12). The halo exchange result, subsequently available in  $bufIn$ , is then copied back to  $U$  (lines 29 – 30).

An important observation of the code in Figure 5.5 is that only the arrays  $bufOut$  and  $bufIn$  are accessed for the halo exchange. The array  $bufOut$  is passed as a local buffer in the  $MPI\_Put$  function, and  $bufIn$  is used in  $MPI\_Win\_allocate$  for the allocated window memory to be remotely accessed. The memory locations of  $U$  and  $U_{new}$  are not accessed at all by any MPI RMA function. For RMA race detection, it means that only  $bufOut$  and  $bufIn$  and their aliases have to be analyzed for RMA races. All memory accesses in the stencil kernel  $compute\_1d\_stencil$  can thus be ignored, as they only access  $U$  and  $U_{new}$ , which can never lead to an RMA race. Since those accesses are in the hotspot of the kernel, ignoring them leads to a significantly reduced runtime tool slowdown.

RMA Sanitizer performs three static analyses on the LLVM Intermediate Representation (IR) to find and instrument relevant local memory accesses. The most effective analysis, the *buffer dependence analysis*, identifies variables used in RMA calls and traverses the alias chain (regarding pointer assignments and function calls) to mark and instrument

```

1 void compute_1d_stencil(double *U, double *Unew) {
2   for (int i = 1; i < N-1; i++)
3     Unew[i] = 0.5 * (U[i-1] + U[i+1]);
4 }
5
6 void halo_exchange(double* bufOut, MPI_Win win, int myrank, int nrank) {
7   MPI_Win_fence(win); // synchronize
8   // write halo data to left and right neighbor
9   MPI_Put(&bufOut[0], 1, MPI_DOUBLE, (myrank-1+nrank) % nrank, ..., win); // left
10  MPI_Put(&bufOut[1], 1, MPI_DOUBLE, (myrank+1) % nrank, ..., win); // right
11  MPI_Win_fence(win); // synchronize
12 }
13
14 int main() { // N: number of elements per process
15   double *U = malloc(N*sizeof(double));
16   double *Unew = malloc(N*sizeof(double));
17   double *bufOut = malloc(2*sizeof(double));
18   double *bufIn, *tmp;
19
20   // init MPI and data (omitted here)
21   MPI_Win win;
22   MPI_Win_allocate(2 * sizeof(double), ..., &bufIn, &win); // for halo exchange
23
24   // stencil loop
25   for (int iter = 0; iter < num_iters; iter++) {
26     bufOut[0] = U[1]; // copy to buffer for halo exchange
27     bufOut[1] = U[N-2];
28     halo_exchange(bufOut, win, myrank, nrank);
29     U[0] = bufIn[0]; // copy from buffer in halo cells
30     U[N-1] = bufIn[1];
31     compute_1d_stencil(U, Unew);
32     tmp = U; U = Unew; Unew = tmp; // pointer swap
33   } // ...
34 }

```

Figure 5.5: 1D-stencil exchange in MPI RMA using *MPI\_Put* and fences. The relevant local memory accesses for RMA race detection are underlined. The associated MPI RMA calls are in boldface. Adapted from [92].

them. As discussed for the example in Figure 5.5, this may significantly reduce the amount of instrumented memory accesses. The *remote access type analysis* checks if a program exclusively performs either remote reads or remote writes to avoid the instrumentation of some local memory reads. The *cluster analysis* avoids superfluous instrumentation of read and write accesses within the same basic block. In the following, the ideas of those analyses will be summarized shortly.

### Buffer Dependence Analysis

In RMA race detection, variables get relevant in two cases: First, if a pointer variable is passed (or returned) as a parameter for the remotely accessible buffer to an RMA allocation routine (such as *MPI\_Win\_allocate* or *shmem\_malloc*), then the variable is relevant for race detection, particularly for remote races. Second, if a pointer variable is passed as a local buffer to an RMA operation routine, it is also relevant for race detection, particularly for local buffer races.

## 5 Scalable Race Detection for RMA Programs

| Source Code                                 | LLVM IR   |
|---|---|
| assume: int* otherbuf, int* <u>localbuf</u> | assume: %0 = otherbuf, %1 = localbuf  |
| otherbuf[0] = 42;                           | %2 = getelementptr i32, ptr %0, i64 0<br>store i32 42, ptr %2                               |
| <u>localbuf</u> [1] = 1337;                 | <b>%3 = getelementptr i32, ptr %1, i64 1</b><br><b>store i32 1337, ptr %3</b>               |
| MPI_Put(& <u>localbuf</u> [1], ..., win)    | %4 = load i32, ptr @win<br><b>%5 = call i32 @MPI_Put(ptr %3, ..., i32 %4)</b>               |
| printf("%d", <u>localbuf</u> [1]);          | <b>%6 = load i32, ptr %3</b><br>%7 = call @printf(ptr @.str, i32 %6)                        |
| <u>localbuf</u> [1] = 0;                    | <b>store i32 0, ptr %3</b>  |
| my_func( <u>localbuf</u> , otherbuf);       | <b>%8 = load ptr, ptr %1</b><br>%9 = load ptr, ptr %0<br>call void @my_func(ptr %8, ptr %9) |

Figure 5.6: Buffer dependence analysis example for local buffer access of MPI\_Put in LLVM IR. The relevant variables are underlined and the bold load/store instructions are the relevant memory accesses. Adapted from [92].

The buffer dependence analysis finds the relevant pointer variables  $p$  and first puts them into an allowlist. Further, for each such pointer  $p$ , a worklist algorithm iteratively searches all pointer assignments or function calls that *use*  $p$  and all pointer assignments that  $p$  itself *uses*. This effectively traverses the use-def and def-use chains of  $p$  to find relevant aliases. Finally, it adds all found corresponding pointer variables also to the allowlist. All load and store instructions to variables in the allowlist are subsequently instrumented.

The algorithm idea is also shown in Figure 5.6 where the pointer to *localbuf* is passed in as %3 to *MPI\_Put* in LLVM IR. All other pointer variables that *use* %3 in their definition, such as %6, and that %3 *uses* in its definition, such as %1, are added to the allowlist. The newly added pointer variables are then analyzed recursively by the worklist algorithm. The analysis also works across function calls, i.e., interprocedural and translation units. Also, the analysis can limit the traversal of the pointer alias chain to a maximum depth to avoid adding aliasing pointers that are “far away”. Details on that are elaborated in [92].

The buffer dependence analysis is conservative, i.e., it may instrument more memory accesses than required. For example, it does not consider the program’s control flow and does not consider whether just a single element of an array or the whole array is accessed. This avoids missing relevant memory access while still being applicable with low static analysis overhead. There might be external function calls (in non-instrumented libraries), e.g., *memcpy*, that may introduce pointer aliases. The buffer dependency analysis cannot detect those which may lead to false negatives in the RMA race detection. Further, calls to function pointers that cannot be statically resolved at runtime might lead to non-detected relevant memory accesses. Both effects on the detection accuracy are further discussed in the evaluation in Chapter 6.

## Remote Access Type Analysis

Depending on the type of remote memory accesses in a program, some load instructions do not have to be instrumented: If an RMA program exclusively performs remote reads, e.g., *MPI\_Get*, then the remote memory regions will only be read from remote. Thus, a local memory read to the remote region can never lead to race. Conversely, if an RMA program exclusively performs remote write calls, e.g., *MPI\_Put*, then the local buffer accesses are only read calls, so local memory reads to the buffer cannot lead to a race and do not have to be instrumented. The remote access type analysis implements this analysis as an extension of the buffer dependence analysis. It does not affect the detection accuracy since it only removes load accesses that are not of relevance.

In the example in Figure 5.6, assuming that *MPI\_Put* is the only RMA operation in the program, it only accesses *localbuf* in a reading fashion. Thus, the *printf* access, which reads from *localbuf*, would not have to be instrumented, as only local memory writes could conflict. As discussed in [92], this optimization only negligibly reduces the overhead because buffers and window memory are often both read and written in many applications, so the optimization cannot be applied.

## Cluster Analysis

The cluster analysis aggregates repetitive accesses to the same memory location within the same basic block. A basic block has only a single entry and a single exit in terms of control flow. Thus, if there are two store instructions to the same memory location, only one of the store instructions has to be instrumented. If there is both a load and a store instruction to the same memory location, then only the (stronger) store instruction has to be instrumented.

Calls to RMA routines (and calls to functions that themselves call RMA routines) within a basic block are optimization barriers: The consistency state of a memory location may change in between, so a memory access before a call to an RMA routine should not be aggregated with a memory access after a call to an RMA routine. The cluster analysis does not change the detection accuracy since it only removes superfluous instrumentation of memory accesses.

## Implementation

All three analyses have been implemented as module passes in LLVM and work on the LLVM IR of the code. The analyses work for all three RMA models: MPI RMA, SHMEM, and GASPI. The buffer dependence and remote access type analysis are first applied to find the relevant pointer variables. Then, the instrumentation itself is performed which is a modified ThreadSanitizer compiler pass: For all load and store instructions

to relevant pointer variables, a call to `__tsan_read` and `__tsan_write` is prepended. The cluster analysis finally detects and removes superfluous instrumentation.

An evaluation on proxy applications in [92] shows that the buffer dependence analysis is the most effective: For most applications, it can reduce the amount of instrumented load and store instructions to 10 – 40 % of the total amount of load and store instructions, depending on the amount of aliasing between pointers in the code. The remote access type analysis did not have any effect because in the examined applications, local buffers of RMA operations were also used as window buffers, which prevented any optimization of this analysis. Also, the cluster analysis only reduced the number of instrumented instructions by a few percent. A detailed overhead analysis is provided in [92]. The effects of the static analysis in RMA Sanitizer will also be evaluated in Chapter 6.

### Related Work

Similar to the buffer dependence algorithm presented here, the post-mortem MPI RMA race detector MC-Checker [16] traverses the AST of the source code to find relevant variables for RMA race detection. However, the analysis implemented in MC-Checker is implemented in the Clang frontend, so it can only be applied to C/C++ programs. The analysis presented in this section directly works on the LLVM IR and thus can analyze any program that can be translated to LLVM IR, such as Fortran code.

PARCOACH [2, 106] also reduces the amount of instrumented memory accesses by performing a static analysis on the LLVM IR. The filter approach checks for each function if it contains a call to a relevant RMA routine and if so, it instruments all subsequent memory access in that function. This analysis works only intraprocedural and misses races with local memory accesses that occur across functions that themselves do not invoke any RMA routine.

### 5.3.2 Synchronization and Consistency Tracking

For race detection, in addition to local memory accesses, the synchronization between processes and the consistency state of RMA operations have to be tracked. For the interception of RMA routines and the exchange of relevant data between processes, RMA Sanitizer uses the infrastructure of P<sup>N</sup>MPI [85], GTI [37], and MUST [36] that has been described in detail in Section 3.4.

Figure 5.7 illustrates the tracking architecture by the example of two processes. An additional tool thread is created for each application process that exchanges relevant information with other tool threads during runtime. The profiling interfaces of MPI [28, §15.2], SHMEM [19, §10], and GASPI [27, §14.2] are used to intercept the RMA routines called in the application at runtime with P<sup>N</sup>MPI which has been extended to work with SHMEM and GASPI. The information on the RMA operations is forwarded to the tool

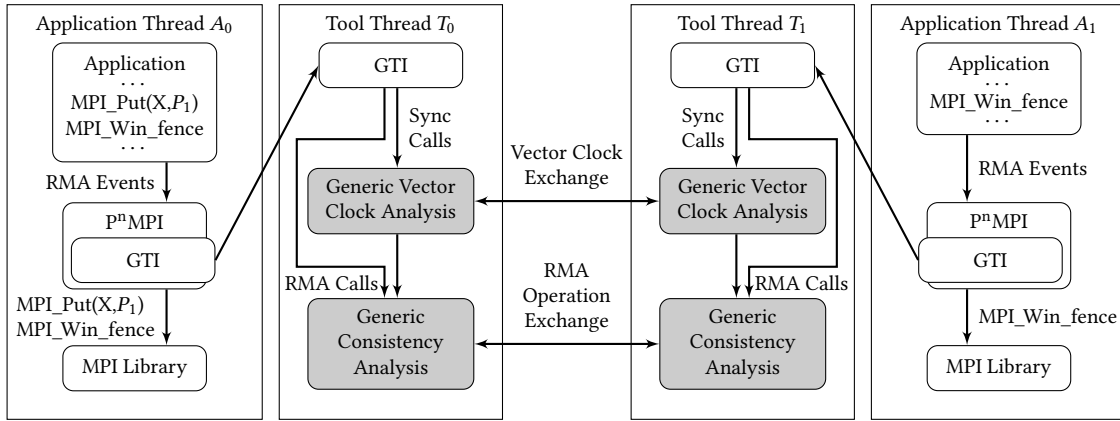


Figure 5.7: Synchronization and consistency tracking architecture of RMASanitizer, exemplified for two processes. For each application process, a tool thread is spawned that tracks the synchronization with vector clocks and the consistency state of the RMA operations. The tool threads of the processes exchange information about synchronization and ongoing RMA operations among each other.

threads via GTI, as explained in detail Section 3.4. In the example, the *MPI\_Put* and *MPI\_Win\_fence* of  $A_0$  are forwarded as events using GTI to the tool thread  $T_0$ .

In the tool threads, two separate analysis modules capture the information relevant to race detection: The *generic vector clock analysis* is the module described in Section 3.4. It receives all calls that establish process synchronization, e.g., *MPI\_Barrier*, and updates the vector clock correspondingly. The *generic consistency analysis* receives all calls relevant for the consistency of RMA operations, i.e., (1) all RMA operation calls and (2) all RMA completion calls. It abstracts the concrete routines of the RMA programming models to the set of generalized consistency events presented in the consistency model in Section 4.1. Both modules exchange relevant information with tool threads from other processes using a GTI communication channel.

At each tool thread  $T_i$ , the generic consistency analysis manages the consistency of (1) all local buffer accesses *issued by*  $P_i$  and (2) all remote accesses *addressed to*  $P_i$ . In the example, the *MPI\_Put* call at the origin  $A_0$  is first forwarded to the local tool thread  $T_0$ . Then, the consistency module in  $T_0$  tracks the consistency status of the local buffer access of *MPI\_Put* locally. The information about the remote write of *MPI\_Put* to  $P_1$  is forwarded from  $T_0$  to  $T_1$ .  $T_1$  then registers the remote *MPI\_Put* received from  $T_1$  internally and tracks its completion state.

The target of an RMA operation must also know about the operation's completion state. Thus, RMA completion calls also have to be intercepted and tracked. The implementation distinguishes two kinds of completion: In active target completion, the target process itself is aware of the completion without requiring knowledge of the origin, e.g., for collective flush operations such as *MPI\_Win\_fence*. In Figure 5.7, the *MPI\_Win\_fence* call at  $A_1$  lets tool thread  $T_1$  know that the previously received *MPI\_Put* of  $P_0$  is completed. In

## 5 Scalable Race Detection for RMA Programs

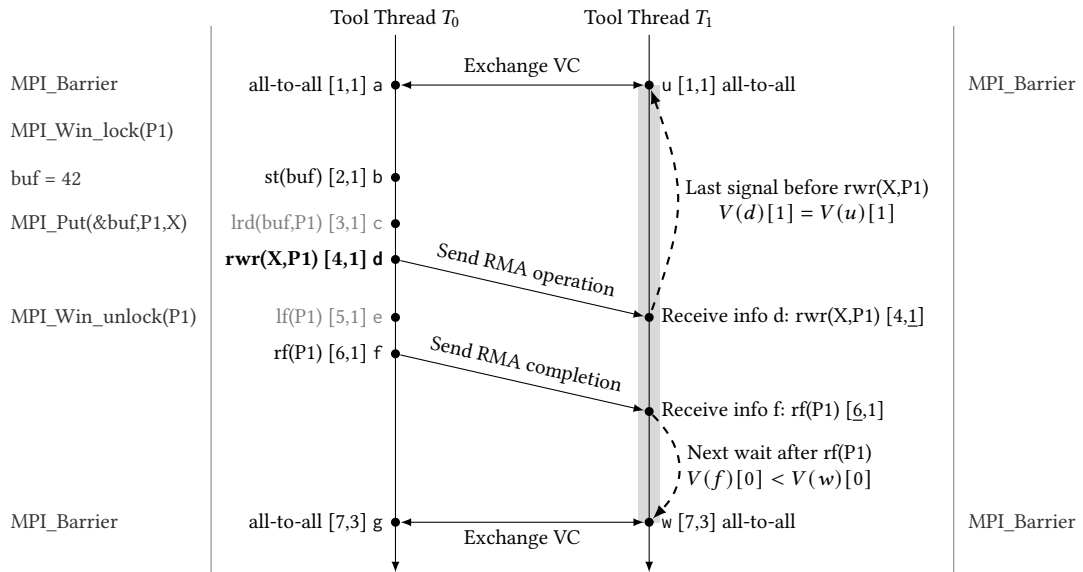


Figure 5.8: Exchange of RMA operation information between tool threads for the example of Figure 4.4. The information of the remote write *rwr* and the remote flush *rf* is sent from tool thread  $T_0$  to tool thread  $T_1$ . Based on the vector clocks of the events, tool thread  $T_1$  can determine the last signal and next wait to get the remote concurrent region of  $(d, f)$ , gray-shaded in the figure. Adapted from [88, 93].

passive target completion, such as remote flushes with *MPI\_Win\_flush*, only the origin is aware of the completion, so the origin has to send information about the completion to the target. The same is true for notify-wait synchronization, in which metadata about the notification is sent from the notifying origin to the waiting target.

In summary, sending metadata about RMA operations and completion from the origin to the target is required to make the analysis work. The remote accesses issued from the origin have to be analyzed for races with (1) local memory accesses at the target and (2) remote accesses from other origins. The only defined place where all those accesses converge is the target process, so forwarding all remote accesses to the target is a natural choice for a distributed analysis.

### 5.3.3 Concurrent Region Detection

RMASanitizer relies on the concurrent region detection as outlined in the race detection model in Section 4.2. For that, the implementation tracks the state of the local buffer and remote accesses at runtime, which is illustrated in the following.

For local buffer accesses, the concurrent region starts with the call to the RMA routine and ends with a corresponding completion call. This detection can run locally on each process and only requires storing and managing all currently open local buffer accesses.

Remote access tracking is more complicated as the target has to know about the remote concurrent region to check for data races. Figure 5.8 shows how this exchange of remote access events works: The *rwr* of *MPI\_Put* is forwarded with its vector clock from  $T_0$  to  $T_1$ . When  $T_1$  receives the information about the *rwr* event, it can use the attached vector clock  $V(d)$  of the event  $d = rwr$  to find the last signal before *rwr* which is the begin of the remote concurrent region. At that point, the remote access has the state *initiated* and is stored for later processing. When  $T_0$  encounters the *rf* event of *MPI\_Win\_unlock*, it also sends the event to the tool thread  $T_1$ . When  $T_1$  receives the *rf* event, it knows about the completion of the previous *rwr* event and changes its state to *completed* for later processing. Lastly, when tool thread  $T_1$  synchronizes with  $T_0$ , i.e.,  $T_1$  has to wait for a signal from  $T_0$ , this is the latest point where the memory effect of *rwr* may occur and therefore denotes the end of the remote concurrent region of *rwr*. In Figure 5.8,  $T_1$  knows with event  $w$  that the remote concurrent region of *rwr* ended.

To make the detection work as described, it is required that the communication channel between tool thread  $T_0$  and  $T_1$  does not allow message overtaking. As the used underlying GTI communication protocol is MPI point-to-point messaging, this property is ensured. Also, the vector clock exchange stalls the application execution at certain points to ensure that the information is up-to-date. At a process synchronization point, the process expecting to receive a vector clock from another process has to stall the application's execution until this vector clock arrives. This has the side effect that all relevant information on previously issued RMA completion calls will also be available.

### 5.3.4 Concurrent Regions as ThreadSanitizer Fibers

The presented concurrent region concept maps to the fibers concept in ThreadSanitizer, as also discussed in [45, 90, 93]. The local buffer and remote accesses of RMA operations are annotated with a corresponding synchronization state to TSan fibers. As discussed in Section 5.3.3, the tracked happened-before order  $\xrightarrow{hb}$  and the consistency order  $\xrightarrow{co}$  can be used to detect the start and end of the concurrent region using vector clocks. The vector clock mechanism used for the concurrent region detection in RMA Sanitizer is completely independent of the vector clock mechanism used in TSan for the race detection in shared memory. Therefore, the main idea is to annotate the detected concurrent region in TSan to make it aware of the concurrent accesses. This is achieved by storing synchronization clocks in TSan as the potential beginning of concurrent regions such that TSan can perform the corresponding race detection. The details of this mechanism are discussed in the following for local buffer and remote accesses.

#### Local Buffer Accesses

For a local buffer access  $l$ , the beginning of the concurrent region is the corresponding RMA call, and the end is a matching completion call  $c$ . TSan has to be made aware

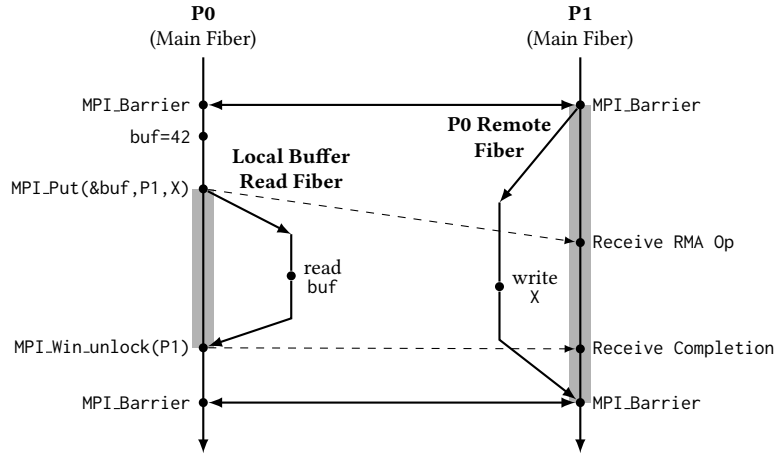


Figure 5.9: Concurrent regions of an `MPI_Put` annotated as fibers.  $P_0$  performs the `MPI_Put` whose local buffer read concurrent region is annotated in an additional fiber at  $P_0$ . The remote write concurrent region is annotated in an additional fiber at  $P_1$ . Any other conflicting access in the gray-shaded area will make TSan report a data race.

that the memory access of  $l$  may have occurred at any time between the beginning and the end of the concurrent region. Therefore, whenever a process calls an RMA routine, RMA Sanitizer calls `AnnotateHappensBefore(handle)` to TSan with a unique identifier `handle`. It conserves the process' synchronization state when the RMA operation started in a synchronization clock with the identifier `handle`.

When the matching completion call  $c$  is detected, RMA Sanitizer creates a new fiber that inherits the synchronization state of the beginning of the concurrent region by calling `AnnotateHappensAfter(handle)`, where `handle` is the identifier of the previously stored synchronization clock for the local buffer access  $l$ . Then, the actual memory access of  $l$  is annotated to that fiber. With this annotated concurrent region, TSan effectively detects races with (1) local memory accesses that occur during the concurrent region and (2) annotated accesses from other concurrent regions. In Figure 5.9, the local concurrent region of an `MPI_Put` is correspondingly annotated as “local buffer read fiber”. To avoid creating a new fiber on every new local buffer access (which would increase the vector clock size significantly in TSan), a pool of fibers is used.

### Remote Accesses

For a remote access  $r$ , the remote concurrent region begins with the last signal before  $r$ , i.e.,  $LS_{t \rightarrow o}(r)$ , and ends with the next wait after RMA completion event  $c$ , i.e.,  $NW_{t \rightarrow o}(c)$ . Different from a local buffer access, the beginning of a remote concurrent region could be potentially *any* synchronization event between processes, e.g., any `MPI_Barrier` call preceding  $r$  in the execution. Therefore, on each process synchronization event, RMA Sanitizer conserves the current TSan synchronization state with `AnnotateHappensBefore(handle)` in a synchronization clock. It stores all of those handles

```

1: procedure AnnotateRemoteConcurrentRegion(rmaOp)
2:   ▶ Store main fiber in variable to switch back later
3:   mainFiber = __tsan_get_current_fiber()
4:   ▶ Create new fiber and switch to it to annotate the operation
5:   opFiber = getRMAFiberForOrigin(rmaOp.origin)
6:   __tsan_switch_to_fiber(opFiber)
7:   ▶ Set synchronization state of fiber to begin of remote concurrent region (last signal)
8:   AnnotateHappensAfter(lastSignal(rmaOp))
9:   ▶ Annotate actual memory accesses to fiber
10:  AnnotateMemoryAccesses(rmaOp)
11:  ▶ Switch back to main fiber, ensure that subsequent memory accesses do not conflict with rmaOp
12:  AnnotateHappensBefore(tmp)
13:  __tsan_switch_to_fiber(mainFiber)
14:  AnnotateHappensAfter(tmp)

```

Figure 5.10: Annotation of a remote concurrent region of an RMA operation in TSan. The synchronization state of the corresponding fiber is set to the last signal and then, the memory access is annotated to that fiber.

in a *last signal map*  $S$  where the key is the current local vector clock value  $V_i[i]$  and the value is the *handle*.  $S(V_i[i])$  then returns for the given local vector clock value  $V_i[i]$  the handle of the corresponding synchronization clock.

As shown in Figure 5.8, the last signal of an event is unambiguously determined by looking at the corresponding vector clock values. As soon as the remote concurrent region of an access  $r$  ends (which, in Figure 5.9, is the case in the second *MPI\_Barrier* at  $P_1$ ), RMA Sanitizer can annotate the region to a fiber: First, it looks up in the signal map the synchronization clock handle  $S(V(r)[t])$  to find the starting point of the concurrent region from the target’s point of view (in Figure 5.9, the first *MPI\_Barrier* call). Then, it switches to the fiber, sets the synchronization state of the fiber to the handle of  $S(V(r)[t])$  with *AnnotateHappensAfter*, annotates the memory access, and switches back. This effectively leads to the remote concurrent region annotated to the remote fiber of  $P_0$  in the example in Figure 5.9. The annotation workflow is also summarized in pseudocode in Figure 5.10.

All remote concurrent regions of RMA operations from the *same* origin  $P_o$  addressed to the *same* target  $P_t$  are annotated to the *same* fiber at the target. The reason for that is efficiency: Often, there are multiple remote accesses from the same origin. Switching for each access to another fiber would be inefficient. Further, conflicting remote accesses from the same origin often have overlapping remote concurrent regions although they do not race, as shown in Figure 4.7, which could lead to a high number of (falsely) detected races in TSan. Therefore, RMA Sanitizer checks pairs of accesses from the same origin to the same target for data races directly in the consistency analysis module, independent from TSan, by comparing their epoch counters and clock intervals as shown in the race detection algorithm in Figure 4.9.

## 5 Scalable Race Detection for RMA Programs

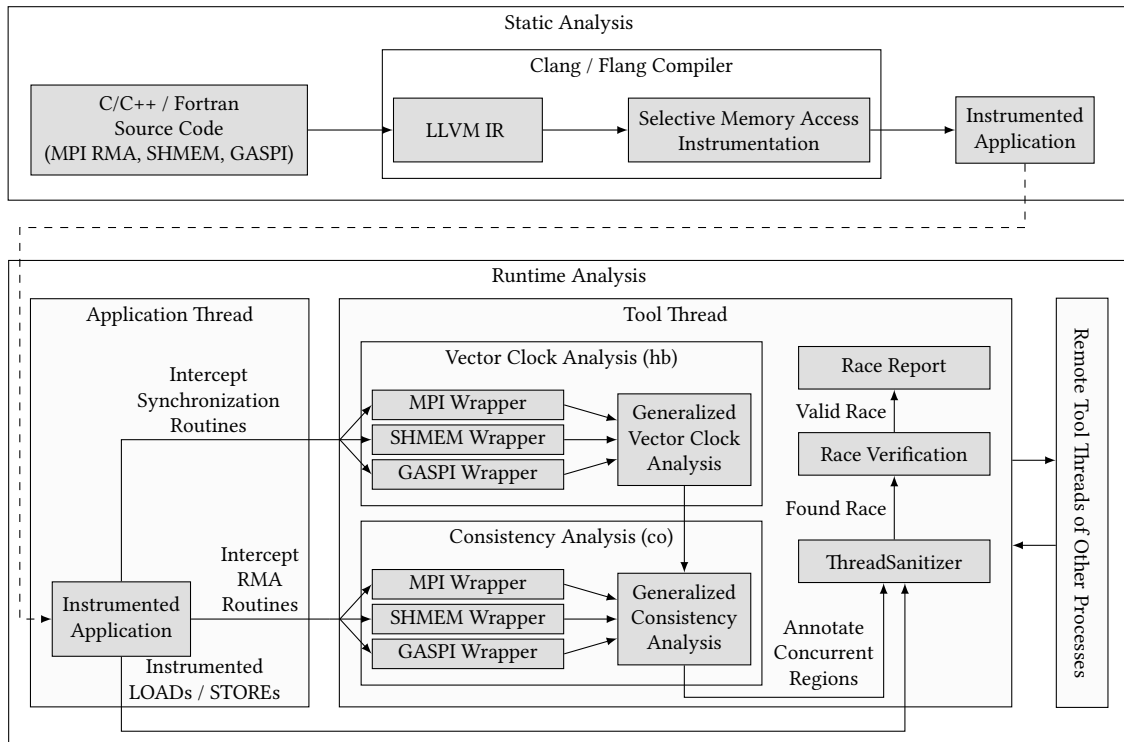


Figure 5.11: RMASanitizer tool workflow. Adapted from [93].

### 5.3.5 Race Detection Workflow

After explaining the components of RMASanitizer, the complete workflow is depicted in detail in Figure 5.11. The source code is first translated to LLVM IR and the relevant memory access instructions are instrumented. Then, the instrumented application is executed and analyzed on-the-fly in a distributed fashion, i.e., a tool thread monitors and analyzes all ongoing RMA accesses at runtime and exchanges this information with other tool threads.

Intercepted synchronization routines are sent to the vector clock analysis module. Wrappers translate the concrete primitives of the RMA programming models to the generalized synchronization primitives for the vector clock exchange. Intercepted RMA communication and completion routines are sent to the consistency analysis module, where wrappers also translate the concrete RMA routines to the abstract consistency primitives of the race detection model. The consistency analysis module also determines the concurrent regions and annotates them to ThreadSanitizer.

The race verification component includes additional checks to verify whether a race reported by ThreadSanitizer is indeed an RMA race and will be described in the following. Also details on the handle tracking and translation, the effect of delayed arrival of RMA operations and an example of the race report format will be discussed in the following.

```

1: procedure VerifyTSanRace(op1, op2)
2:   ▷ If one of the accesses is not a remote access, then it is a valid race.
3:   if not isRemoteAccess(op1) or not isRemoteAccess(op2) then
4:     └ return True
5:   ▷ If both are atomic and compatible (same data type, aligned offsets), this is no race.
6:   if isAtomic(op1) and isAtomic(op2) and datatype(op1) == datatype(op2)
7:     and startAddr(op1) - startAddr(op2) % typeSize(op1) == 0 then
8:       └ return False
9:   ▷ If accesses are externally synchronized (disjoint local concurrent regions), this is no race.
10:  if startClock(op2) > endClock(op1) or startClock(op1) > endClock(op2) then
11:    └ return False
12:  return True

```

Figure 5.12: Verification of a race reported by TSan on two memory operations. The procedure returns true if the detected race is an RMA race, otherwise false.

## Race Verification

The described analysis workflow with the annotation of concurrent regions in ThreadSanitizer only checks the first part of the race detection algorithm described in Section 4.2.3 and also ignores the specificity of RMA atomics. Therefore, any TSan race report must additionally be verified by RMASanitizer if it is indeed an RMA race. Figure 5.12 shows a pseudocode of the race verification that will be described in the following.

First, if at least one of the two accesses is not a remote access, then the detected race is indeed an RMA race. Second, RMA atomics are not annotated as atomic accesses to ThreadSanitizer because otherwise, TSan would treat RMA atomics as compatible with the programming language atomics (in C/C++) which is not the case, leading to false negatives. However, ignoring the atomicity of RMA accesses means that TSan may falsely report races if atomic RMA accesses concurrently access the same memory location. To avoid those false reports, RMASanitizer checks for a race reported by TSan whether the corresponding RMA accesses are atomic and compatible according to RMA semantics.

Third, as discussed in the context of the race detection algorithm, overlapping concurrent regions at the destination are a necessary but not sufficient property of a race, as accesses might still be (1) fence-ordered or (2) externally synchronized. The fence order can only be established between remote accesses from the same origin to the same target. As remote accesses from the same origin are annotated to the same fiber, they cannot lead to a detected race by TSan anyhow. Races between remote accesses from the same origin are directly analyzed by the consistency analysis module. However, external synchronization of accesses leads to false data race reports of TSan. Therefore, RMASanitizer internally verifies ThreadSanitizer's race report by performing a full vector clock comparison of the start and end of the local concurrent region. The check in line 9 of Figure 5.12 corresponds to the check in line 19 of Figure 4.9.

## Tracking and Translating Handles

All three RMA models have specific ways of representing remotely accessible regions and ensuring completion that all have to be captured correctly by the generalized consistency module, e.g., MPI RMA uses windows to represent accessible memory regions and SHMEM uses contexts to associate completion only with certain RMA operations. To cope with all possible scenarios, the consistency module defines three identifiers that are used by the model-specific wrappers in the translation step, namely an RMA operation id, an RMA segment id, and an RMA context id.

The *RMA operation id* uniquely identifies each RMA operation. It allows the state of a particular RMA operation to be modified. MPI RMA allows changing the completion state of an individual RMA operation through request-based RMA operations such as *MPI\_Rget* and *MPI\_Rput*. Correspondingly, the MPI RMA wrapper stores the current state of the MPI request locally and associates it with the RMA operation id. When the MPI request is completed, it signals local completion to the consistency module of that particular RMA operation. In GASPI, individual completion is possible via *gaspi\_read\_notify* and *gaspi\_notify\_waitsome* that is also handled in the GASPI wrapper and passed to the consistency module. SHMEM does not allow for individual completion of RMA operations besides using blocking calls such as *shmem\_put* which immediately completes locally.

The *RMA segment id* is required for MPI RMA and GASPI to identify the affected remotely accessible regions (“windows” in MPI RMA and “segments” in GASPI). In MPI RMA and GASPI, remote accesses at the origin are issued relative to the segment’s base address. When the target receives information about an ongoing RMA operation, it must translate this relative offset to an absolute address. For that, it uses the base address stored together with the RMA segment id. Although SHMEM does not distinguish between segments in its semantics, even then, a remote access issued at the origin to a remote memory address at the target still has to be translated, because address-space layout randomization will make an allocated SHMEM segment still reside at different address locations on different processes.

The *RMA context id* is specifically required for SHMEM and GASPI. In SHMEM, issued RMA operations can be associated with a certain context identifier, e.g., with *shmem\_ctx\_put*, and there are corresponding calls such as *shmem\_ctx\_quiet* that ensure completion of RMA operations associated with that context. A concept with similar semantics provided by GASPI are *queues*. The generic consistency module subsumes both concepts using the RMA context id.

MPI RMA and SHMEM allow identifying process groups with *communicators* and *teams*, respectively. For example, completion in MPI RMA and SHMEM can be established for a certain process group. The wrapper modules translate this model-specific representation of a process group to a list of processes before passing the information to the generic consistency module.

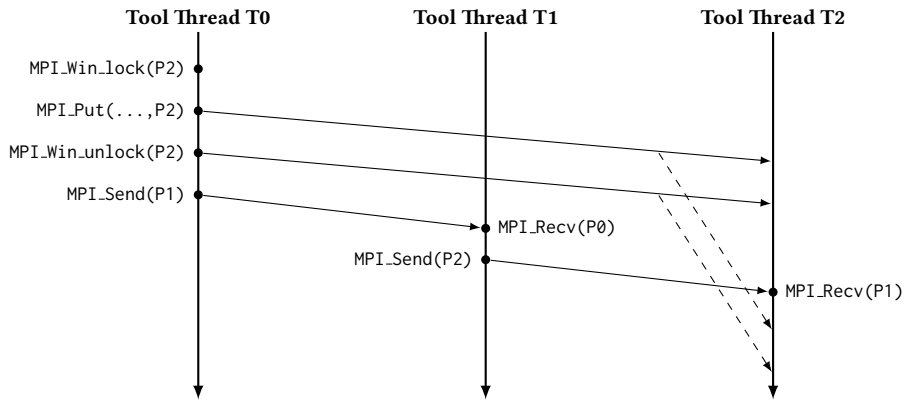


Figure 5.13: Delayed arrival of RMA operation metadata. The information about the ongoing `MPI_Put` and `MPI_Win_unlock` may be delayed (dashed line) such that it arrives after the `MPI_Recv` at  $P_2$ , potentially leading to false positives. Adapted from [88].

### Delayed Arrival of RMA Operations

RMASanitizer relies on a timely reception of RMA operations and completion information at the target to annotate the remote concurrent regions to TSan. As previously described, the vector clock exchange stalls the application execution to receive the corresponding vector clocks, which, as a side effect, also ensures that previously issued RMA operations will be received on time. However, this cannot be ensured when RMA operations are completed through transitive synchronization, as discussed in [88] and shown in Figure 5.13. The information about the remote write of the `MPI_Put` from  $P_0$  to  $P_2$  and its completion in `MPI_Win_unlock` is sent out as a message from the tool thread of  $P_0$  to the tool thread of  $P_2$ . The synchronization from  $P_0$  to  $P_1$  and then from  $P_1$  to  $P_2$  ensures that the `MPI_Win_unlock` at  $P_0$  happens before the `MPI_Recv` at  $P_2$ , so the concurrent region of the `MPI_Put` ends there. However, due to network delays, it might be that the messages with the ongoing `MPI_Put` and `MPI_Win_unlock` are delayed so that the concurrent region is annotated too late, potentially leading to false positives as RMASanitizer assumes the concurrent region to be longer than it actually is.

In practice, this limitation will occur only in rare cases: It requires that the application relies on transitive synchronization for the completion of RMA operations, which is typically unlikely. Even if so, the delay of the RMA operations must be very large such that sending a message from  $P_0$  to  $P_2$  takes longer than sending a message from  $P_0$  to  $P_1$  and then another message from  $P_1$  to  $P_2$ . Therefore, this limitation also did not appear in any of the experiments presented in Chapter 6.

### Race Report Format

When RMASanitizer detects an RMA race, it reports the race to the user. It uses the existing error reporting infrastructure of MUST to generate the report, either in textual or

---

```

1 int main(int argc, char **argv) {
2   int rank, size; MPI_Win win; int *win_base;
3   MPI_Init(&argc, &argv);
4   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
5   MPI_Win_allocate(sizeof(int), sizeof(int), MPI_INFO_NULL, MPI_COMM_WORLD,
6                   &win_base, &win);
7   win_base[0] = 42;
8
9   MPI_Barrier(MPI_COMM_WORLD);
10  if (rank == 0) {
11    int value = 1;
12    MPI_Win_lock(MPI_LOCK_EXCLUSIVE, 1, 0, win);
13    MPI_Put(&value, 1, MPI_INT, 1, 0, 1, MPI_INT, win); // write to win_base[0] at rank 1
14    MPI_Win_unlock(1, win);
15  }
16  if (rank == 1) {
17    printf("win_base[0] is %d\n", win_base[0]); // race
18  }
19  MPI_Barrier(MPI_COMM_WORLD);
20  [...]

```

---

```

1 ===== RMA Sanitizer Output =====
2 Remote data race at rank 1 between a write of size 4 at reference 1 from rank 0
3 and a previous read of size 4 at reference 2 from rank 1.
4 Reference 1: MPI_Put@main, race-example.c:13@rank 0
5 Reference 2: main, race-example.c:17@rank 1
6
7 Concurrent region of reference 1 started at reference 3 and ended at reference 4.
8 Reference 3: MPI_Barrier@main, race-example.c:9@rank 1
9 Reference 4: MPI_Barrier@main, race-example.c:19@rank 1

```

---

Figure 5.14: MPI RMA remote race example with RMA Sanitizer output.

in HTML form. An example output of an RMA race in text form is shown in Figure 5.14 together with the corresponding source code. The report lists the kind of race (local buffer or remote race) and the conflicting accesses, including their stack traces and originating ranks. The beginning and end of the concurrent region of the conflicting RMA operation are also shown as optional debugging help.

### 5.3.6 Alternative Access Tracking with Interval Skip Lists

RMA Sanitizer relies partly on TSan for the race detection which uses shadow memory to keep track of the status of the application memory. The advantage of shadow memory is the access time of  $O(1)$  because finding the shadow memory cell for a given application memory address only requires adding an offset. The disadvantage is the (constant) memory overhead of 2x to 4x compared to the memory consumption without TSan.

Interval Skip Lists (ISLs) [34] have been used successfully in UPC-Thrille [73] for race detection, mainly due to their ability to efficiently search for overlapping intervals and their simple implementation. A skip list is a linked list of elements with additional levels that work as fast lanes for efficient inserts and search queries, effectively reducing the expected time from  $O(n)$  to  $O(\log n)$  for insertion, search, and deletion. The same principle is applied to intervals in the Interval Skip List (ISL). For an interval  $I = [a, b]$ ,

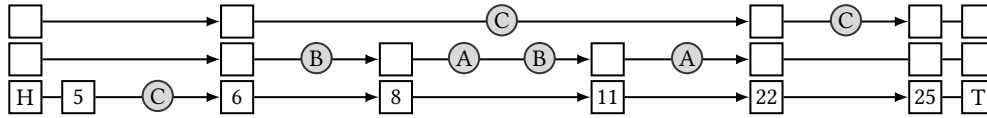


Figure 5.15: Interval Skip List for intervals A [8, 22], B [6, 11], and C [5, 25]. Taken from [47].

the start point  $a$  and the endpoint  $b$  are added as nodes to the list and edges between those nodes are correspondingly marked with the interval  $I$ . Figure 5.15 shows an example of such an ISL. While the ISL has a higher access complexity of  $O(\log n)$  than shadow memory with  $O(1)$ , the required space of the ISL depends on the number of memory accesses compared to the constant (high) memory overhead of TSan.

The bachelor thesis of Sem Klauke [47] implements an alternative data race detection in RMASanitizer that uses ISLs as a replacement for ThreadSanitizer’s shadow memory approach, i.e., each process locally manages an ISL with all the local and (annotated) remote memory accesses. The vector clock analysis and consistency module stay the same; the only difference is that the annotation of the concurrent region is processed by the ISL instead of TSan. For each memory access, an interval covering the start and end address is added to the list. Further, each interval stores additional metadata, such as the type of access and its concurrent region, in the form of vector clocks. On every new memory access added to the list, any overlapping interval in the ISL is analyzed for conflicts and for concurrency by comparing the vector clocks of both accesses.

In experiments performed by Klauke [47], the ISL implementation shows orders of magnitudes higher slowdowns than the TSan implementation. The main reason is that applications perform many local memory accesses, which leads to a massive amount of data that has to be processed by the ISL infrastructure. To avoid polluting the ISL with single intervals for each local memory access, consecutive memory accesses are aggregated. This processing, however, takes time. Further, in the current implementation, each local memory access travels through the whole GTI infrastructure from the application thread to the tool thread where the ISL is managed, which introduces additional overhead.

The memory consumption in the experiments was sometimes lower and sometimes higher than the TSan variant. The higher the number of memory accesses, the higher the memory consumption in the ISL, while it stays constant with TSan. This is because TSan evicts older memory accesses in the shadow state when new memory accesses come in. Another problem with the ISL approach is the question when inserted memory accesses become irrelevant and how to find them. In the implementation of Klauke [47], whenever there is a collective flush operation such as `MPI_Win_fence` in the program, the whole ISL is cleared because accesses after the collective flush cannot be in conflict with accesses before the collective flush. However, this does not hold if only a subset of the processes participate in the collective flush operation. Further, this optimization does not have an effect on programs that do not use collective flushes at all.

Since the slowdown with the ISL approach is significantly higher and the memory consumption is typically similar to or higher than with TSan, the TSan approach is the preferred variant used in RMA Sanitizer.

### 5.3.7 Limitations

RMA Sanitizer inherits the limitations of the synchronization and the consistency model discussed in Section 3.3.5 and Section 4.3. This section discusses implementation-specific limitations of RMA Sanitizer that could be addressed in future work.

MPI RMA and SHMEM allow processes that physically share memory, e.g., since they are located on the same node, to access remote memory segments directly via plain load and store accesses. MPI RMA defines for that a routine `MPI_Win_allocate_shared` [28, §12.2.3] to allocate a *shared window*. If all processes of the given communicator are in the same shared memory domain, it allocates a window as a shared memory segment across processes and allows direct access via loads and stores to the segment by any process. The consistency of the load and store accesses to the shared memory segment still has to be ensured by RMA completion routines as for usual RMA operations. Similarly, SHMEM provides a routine `shmem_ptr` which returns for a given symmetric address a local pointer that can be used to directly access a remote memory location of another process, if possible. Currently, RMA Sanitizer solely relies on the interception of RMA routines via the profiling interfaces to get aware of RMA operations, so direct load and store accesses to remote memory locations are currently ignored and races might be missed. One way to incorporate those accesses would be to rely on the instrumentation of ThreadSanitizer that already instruments all plain load and store accesses: The accesses to remote memory locations could be separated from local memory accesses based on the accessed address range and then reported to the consistency module of RMA Sanitizer.

RMA Sanitizer currently only supports single-threaded RMA programs. The execution with multiple threads per process is possible, but may lead to false negatives as RMA Sanitizer interprets multithreaded executions within a process as if they were serialized in some sequential order. The main reason is that the vector clock in RMA Sanitizer only stores a single integer for each process. If a process consists of multiple threads that concurrently call synchronization-relevant routines, then a call by any thread increments the local vector clock entry. The information that those calls within a process are multithreaded is lost. The effect of that will be further elaborated in Section 6.1. A future extension to hybrid vector clocks, as discussed in Section 3.6, could avoid this problem.

The currently used compiler instrumentation of local memory accesses requires the availability of the program's source code. In particular, the static analysis in RMA Sanitizer relies on scanning the source code to detect irrelevant local memory accesses. However, if an analysis on an application in binary form is desired, there is no restriction in coupling the RMA Sanitizer analysis with (dynamic) binary instrumentation frameworks such as Valgrind [66], but at the cost of significant higher runtime overhead.

## 5.4 Results and Discussion

This chapter presents RMASanitizer, an on-the-fly data race detector for RMA programs. Its primary goal is to achieve high data race detection accuracy while keeping the overhead manageable in real-world applications. RMASanitizer implements the race detection model defined in Chapter 4 by utilizing the MUST and GTI infrastructure. For synchronization tracking, it uses the vector clock exchange model described in Chapter 3. The consistency tracking implements the concurrent region tracking to understand the completion state of ongoing RMA operations.

RMASanitizer is the first on-the-fly detector for RMA programs that supports different synchronization mechanisms (collective flushes, remote flushes, notify-wait) and programming models (MPI RMA, SHMEM, GASPI). The related on-the-fly RMA race detector implemented in PARCOACH [2] only works for MPI RMA programs that collectively complete and synchronize, so any mechanism of individual completion (besides experimental support for *MPI\_Win\_flush*) is not supported. The issue with post-mortem approaches such as MC-Checker, MC-CChecker, or the approach of Krzikalla [49] is that they can only detect the RMA race after the application has terminated. For long-running applications, the trace size might get unmanageable large.

RMASanitizer combines a set of static analysis passes to only instrument those load and store instructions of the program that are relevant for the race detection, which is a significant optimization to achieve a scalable analysis, as also discussed in related work [2, 16, 106]. It works on the LLVM IR level and is therefore compatible with both C/C++ and Fortran. Previous approaches are either limited to C/C++ programs or use intraprocedural analyses, which miss many relevant memory accesses.

RMASanitizer partly relies on ThreadSanitizer for the race detection, which is a scalable race detector for shared-memory programs using shadow memory to track the status of individual memory locations. RMASanitizer annotates the detected concurrent regions using the ThreadSanitizer annotation API as fibers. As not all semantics of RMA models, such as RMA atomics, can be annotated and modeled in ThreadSanitizer, RMASanitizer includes an additional race verification that checks if a race reported by ThreadSanitizer is indeed a race.

By relying on a generalized race detection model, RMASanitizer is capable of supporting various variants of RMA models. Currently, it supports MPI RMA, SHMEM, and GASPI. Its architecture is designed so that adding support for another RMA model only requires adding wrappers from the concrete RMA routines to the corresponding abstract primitives of the race detection model. The remaining part of the infrastructure can be reused. As the integration of MPI RMA, SHMEM, and GASPI has shown, specific concepts might still require further extension of the generalized consistency module in practice. For example, SHMEM has the concept of communication contexts, which is semantically different from the MPI RMA concept of a window. Therefore, both had to be integrated as separate concepts in the generalized consistency module of RMASanitizer.

## *5 Scalable Race Detection for RMA Programs*

Like the RMA race detection model, RMA Sanitizer only supports an accurate data race detection in single-threaded executions. However, an execution of multithreaded programs is in principle possible, but any RMA race occurring due to using multithreading will not be reliably detected. An evaluation of RMA Sanitizer on RMA races in hybrid programs is discussed in Chapter 6.

## 6 Classification Quality and Overhead Analysis of RMA Race Detection

Correctness checking tools should aid users in finding bugs in their programs. Therefore, the primary quality metric of a tool is its ability to detect bugs in applications reliably, often also referred to as *classification quality*. In the best case, a tool can correctly detect all bugs in a program without any falsely reported bug. However, in practice, this is nearly impossible to achieve since the interaction in (parallel) programs is complex to analyze, especially due to the various ways of establishing synchronization and consistency. Quantifying the detection accuracy is, therefore, required to estimate the quality of a tool. Further, a correctness checking tool that requires hours or days to analyze a program that runs for just a few minutes may not be helpful at all. Therefore, the overhead of a tool should also be analyzed. This chapter presents both an analysis of the classification quality and the introduced overhead of RMA Sanitizer.

To assess the classification quality of correctness checking tools, benchmark suites consisting of simple test codes with injected bugs and bug-free codes have been developed. The test codes are used to verify how many test cases are classified correctly to determine the tools' error detection accuracy. RMA Sanitizer is evaluated with the classification quality benchmark suite *RMA Race Bench* [91] containing about 100 RMA race test cases (with and without race) for each MPI RMA, SHMEM, and GASPI. The results of RMA Sanitizer on *RMA Race Bench* are also compared to the state-of-the-art RMA race detectors implemented in PARCOACH [2, 82, 106].

The introduced overhead of a correctness checking tool is also a fundamental property to be evaluated. RMA Sanitizer has been tested on different RMA proxy applications written in MPI RMA, SHMEM, and GASPI on large-scale runs with up to 768 processes to evaluate its real-world applicability. This includes a detailed analysis of its scalability behavior and an overhead breakdown of the main overhead contributors. Further, the overhead of RMA Sanitizer is compared to the overhead of the dynamic RMA race detector implemented in PARCOACH [2, 106].

This chapter is structured as follows: Section 6.1 analyzes the classification quality of RMA Sanitizer on the classification quality benchmark suite *RMA Race Bench*. I have initially published *RMA Race Bench* in [91]. Section 6.2 presents the overhead results of RMA Sanitizer on different RMA proxy applications and a comparison to PARCOACH. The overhead results of RMA Sanitizer on the proxy applications have been previously published in [93] but were re-evaluated for this thesis.

## 6.1 Classification Quality with RMARaceBench

The classification quality of a correctness tool can be analyzed by defining a set of test cases with and without errors and running the tool on the tests to check if it correctly classifies a test as erroneous or error-free. For RMA race detectors, the test set correspondingly consists of test cases with and without RMA races. The evaluation of a tool corresponds to the evaluation of a binary classifier with the usual classes true positive (TP), true negative (TN), false positive (FP), and false negative (FN). This section presents the classification quality benchmark suite RMARaceBench [91] and the evaluation results on RMA Sanitizer and other RMA race detectors.

RMARaceBench [91] is a semantically-driven microbenchmark suite consisting of synthetic RMA race test cases to evaluate the classification quality of RMA race detectors. It provides test cases written in MPI RMA, SHMEM, and GASPI. The property “semantically driven” means that the test cases cover all kinds of races (local buffer races and remote races) due to the wrong usage of completion and synchronization mechanisms in the three RMA models. The structure of RMARaceBench is similar to other classification quality benchmarks such as DataRaceBench [61] and MPI-Corrbench [59]. The code and infrastructure of RMARaceBench<sup>1</sup> is publicly available under a BSD 3-clause license. In the following, the design of RMARaceBench and the test case categories will be explained in detail.

### 6.1.1 Methodology

The main goal of RMARaceBench is to challenge the detection coverage of an RMA race detector because RMA models have various completion and synchronization concepts that should all be supported by a tool. For all the different RMA communication variants (remote read and write, atomic accesses) and completion modes (active target completion, passive target completion, notify-wait) discussed in detail in Section 2.2.3, RMARaceBench provides corresponding test cases with and without races. Further, there are tests relying on different process synchronization constructs (barriers, send/receive pairs, locks). The tests also cover the interaction with hybrid parallelism by combining the RMA models with OpenMP.

Each test case in RMARaceBench is a self-contained C source code file that can be compiled and executed, making it suitable for analysis with static and dynamic analysis tools. C has been chosen as a programming language because all three RMA models provide C bindings. The test cases are generated using a Python infrastructure that relies on Jinja<sup>2</sup> templating to avoid writing the same boilerplate code for every test. Meta information is also encoded in the test cases, such as the kind of race and the affected source code lines, as exemplified in Figure 6.1.

---

<sup>1</sup><https://github.com/RWTH-HPC/RMARaceBench>

<sup>2</sup><https://jinja.palletsprojects.com>

```

1  /* METADATA BEGIN
2  {
3     "RACE_KIND": "remote",
4     "ACCESS_SET": ["rma read", "rma write"],
5     "RACE_PAIR": ["MPI_Get@22", "MPI_Put@26"],
6     "NPROCS": 3,
7     "DESCRIPTION": <omitted here>
8  }
9  METADATA END */
10
11 int main(int argc, char** argv)
12 {
13     int rank, size;
14     MPI_Win win; int* win_base;
15     int value = 0; int target_rank = 1;
16
17     /* boilerplate: initialization (omitted here) */
18
19     MPI_Win_fence(0, win);
20
21     if (rank == 0) {
22         MPI_Get(&value, 1, MPI_INT, target_rank, 0, 1, MPI_INT, win); // CONFLICT
23     }
24     if (rank == 2) {
25         value = 2;
26         MPI_Put(&value, 1, MPI_INT, target_rank, 0, 1, MPI_INT, win); // CONFLICT
27     }
28
29     MPI_Win_fence(0, win);
30
31     /* boilerplate: shutdown (omitted here) */
32
33     return 0;
34 }

```

Figure 6.1: RMARaceBench test case example for the conflict category showing a remote race originating from *MPI\_Get* at rank 0 and *MPI\_Put* at rank 2 both addressing target rank 1. The other test cases of the conflict category have the same structure, with the only difference being that other combinations of RMA and local accesses are tested. Adapted from [91].

RMARaceBench contains test cases with and without races. Any test with a race contains exactly one race and the data race is always observable, i.e., there is no non-determinism leading to different execution paths hiding the race. Any tool (dynamic or static) should be able to observe it. Further, each race can be attributed to two conflicting accesses from exactly two source code lines. This design allows for an unambiguous classification of the tool results. The tests only contain the minimum amount of required boilerplate code and are kept short. While the races are easy to spot in the test cases of RMARaceBench, this may not be true in larger code bases where manual inspection could be complex.

The test cases are grouped into the categories “conflict”, “synchronization”, “atomic”, “hybrid”, and “miscellaneous”, which challenge the different capabilities of the RMA race detectors. The categories will be presented below. The total number of test cases for each category is shown in Table 6.1. A complete list of all test cases is provided in Appendix B.

Table 6.1: Number of incorrect, correct, and total test cases in RMA RaceBench for MPI RMA, SHMEM, and GASPI. Adapted from [91].

| Category        | # Incorrect   # Correct   # Total Test Cases |               |               |
|-----------------|--|---------------|---------------|
|                 | MPI RMA                                      | SHMEM         | GASPI         |
| Conflict        | 26   13   39                                 | 35   14   49  | 32   11   43  |
| Synchronization | 20   17   37                                 | 12   12   24  | 8   6   14    |
| Atomic          | 6   4   10                                   | 5   4   9     | 1   2   3     |
| Hybrid          | 12   10   22                                 | 12   10   22  | 12   10   22  |
| Misc            | 9   9   18                                   | 9   9   18    | 9   9   18    |
| Total           | 73   53   126                                | 73   49   122 | 62   38   100 |

### Conflict Category

When a race detector finds concurrent memory accesses to the same memory location, it must check whether the accesses are indeed conflicting. The *conflict* category systematically tests through different kinds of access combinations to check whether an RMA race detector can correctly identify conflicting accesses. For each access kind combination listed in Table 2.1 for local buffer accesses and listed in Table 2.2 for remote accesses, a test case is generated where those accesses are concurrent. There are 7 combinations for local buffer accesses and 18 combinations for remote accesses, resulting in 25 possible access kind combinations. The local buffer race tests use a code template similar to Figure 2.7, and the remote race tests use a code template similar to Figure 6.1. The code example in Figure 6.1 shows the generated code for the combination (remote read, remote write) for MPI RMA using *MPI\_Get* and *MPI\_Put*.

All three RMA models provide several communication calls with the same access type. For example, MPI RMA has *MPI\_Put* and *MPI\_Rput*, which both execute a remote write, and SHMEM has *shmem\_get* and *shmem\_get\_nbi*, which both execute a remote read. For each of the 25 conflict combinations, only one representative, e.g., *MPI\_Put* for a remote write, is chosen for each access kind to avoid an exceeding number of test cases. Additional variations are included to cover all (remaining) RMA routines that are not considered in the generation of the 25 access kind combinations. There are 39 tests for MPI RMA, 49 tests for SHMEM, and 43 tests for GASPI in total.

### Synchronization Category

An RMA race detector also has to analyze whether conflicting (RMA) accesses are concurrent or not to identify races correctly. For that, it has to understand the completion and synchronization semantics of RMA operations that influence the concurrent regions of RMA operations. Given a conflicting set of (RMA) accesses, the *synchronization* category

systematically tests through completion mechanisms in the RMA models. For each completion mechanism, at least one test case uses it correctly to avoid concurrent conflicting accesses, and at least one test case uses it incorrectly, resulting in concurrent conflicting accesses and thus a data race. Using those test cases, RMA RaceBench can identify which completion and synchronization mechanisms an RMA race detector supports.

The covered completion variants are those described in Section 2.2.3 for MPI RMA (fence, PSCW, lock/unlock, lock-all/unlock-all, flush, flush-local), SHMEM (barrier-all, quiet, fence, wait-until, signal), and GASPI (wait, notify, notify-wait). It covers all mechanisms present in the three RMA models. This leads to 37 test cases for MPI RMA, 24 for SHMEM, and 14 for GASPI. GASPI has significantly fewer test cases than MPI RMA and SHMEM, as it only provides *gasp\_i\_wait* for local completion and a notify-wait synchronization mechanism.

### Atomic Category

As described in Section 2.3.4, RMA atomic operations may avoid data races, but only when they are used correctly. Atomicity between two RMA accesses is only given if (1) both accesses are RMA atomics, (2) both accesses use the same data type, and (3) the accesses are aligned correctly (in terms of byte offsets). The *atomic* category provides test cases with concurrent atomic operations that correctly adhere to the atomicity rules and test cases violating one of the atomicity requirements, leading to a remote race. The number of test cases is 10 for MPI RMA, 9 for SHMEM, and 3 for GASPI. GASPI has no data types and can only run into the alignment problem, hence the small number of test cases.

### Hybrid Category

The *hybrid* category provides test cases that use OpenMP in conjunction with the RMA models, which may lead to more complex race scenarios. As discussed in Section 2.3.5, when using multithreading on the origin side, there may be one thread performing an RMA operation while another thread concurrently accesses the local buffer of the RMA operation in a conflicting way, leading to a local buffer race. On the target side, synchronization with the origin might be established only with a designated single target thread but not with other threads of the target that perform conflicting accesses, leading to remote races. An example of that has been discussed in Figure 2.12.

The tests of the hybrid category cover the major concepts of OpenMP, namely work-sharing constructs, tasking, the single and master construct, tasks, and sections, also combined with barrier constructs, if necessary. For each concept, there is a test case where improper OpenMP synchronization between threads with conflicting accesses leads to a local buffer race or remote race and another test where proper OpenMP synchronization prevents a race between the threads. There are 10 local buffer access tests and 12 remote

access tests for each RMA programming model. The test cases are designed such that a tool has to understand OpenMP parallelism to classify the races correctly.

### Miscellaneous Category

RMASanitizer uses selective local memory access instrumentation based on static analyses to avoid instrumenting memory accesses irrelevant to RMA race detection. As discussed in Section 5.3.1, using the buffer dependence analysis for filtering may lead to false negatives in the race detection in some cases because it might not detect aliases via external functions or function pointers correctly.

In the context of the research work on the static analysis [92], RMARaceBench has been extended with a *miscellaneous* category to stress test the buffer dependence analysis, but also static analysis tools in general. While in all test cases of the other test categories, the whole program code is contained within a single main function, the miscellaneous test cases cover scenarios where RMA races occur between accesses across function boundaries, i.e., interprocedural, and pointer aliases are introduced in different ways. In particular, the test cases cover aliases between buffers introduced via (1) a called function, (2) a called function pointer, and (3) external function calls that generate aliases, e.g., *memcpy*. Further, there are test cases with deeply nested function calls to check whether a race detector can still detect races across functions.

### 6.1.2 Related Benchmark Suites

RMARaceBench is not the first classification quality benchmark suite to test the accuracy of correctness tools for parallel programs. DataRaceBench [61] provides 200 OpenMP test cases with and without data race. The test cases were extracted and adapted from existing test suites, extracted as kernels from real-world applications, or newly created. The benchmark suite covers the whole OpenMP feature set, e.g., worksharing, tasking, and SIMD constructs. Using DataRaceBench, different state-of-the-art OpenMP detectors were compared. DataRaceOnAccelerator [83] focuses on data races in offloading programming using OpenMP, OpenACC, and CUDA. It provides test cases with and without races between the host and accelerator and on the accelerator itself. The test suite comprises about 40 hand-written synthetic test cases per programming model. The Indigo suite [62] includes variations of six basic code patterns focusing on parallel graph applications. The suite systematically mutates those basic code patterns to generate different test variants, also by injecting data races into the code, leading to a set of 1720 test cases.

Several efforts have been made to evaluate the classification quality of correctness tools for MPI applications. MPI-Corrbench [59] consists of 510 synthetic hand-written microbenchmarks, partly extracted from existing regression test suites, and tests through different error patterns such as invalid arguments, deadlock, message races, and resource leaks. It contains correct and incorrect test cases. MPI-Corrbench also provides a few

```

1 int main(int argc, char** argv)
2 {
3     int rank, size;
4     MPI_Win win; int* win_base;
5     int value = 1;
6     int buf = &value;
7
8     /* boilerplate: initialization (omitted here) */
9
10    MPI_Win_fence(0, win);
11
12    /* create alias via memcpy */
13    int* buf_alias;
14    memcpy(&buf_alias, &buf, sizeof(int*));
15
16    if (rank == 0) {
17        // local write of buf
18        MPI_Get(buf, 1, MPI_INT, 1, 0, 1, MPI_INT, win); // CONFLICT
19        // local read of buf_alias, which is an alias of buf
20        printf("*buf_alias is %d\n", *buf_alias); // CONFLICT
21    }
22
23    MPI_Win_fence(0, win);
24
25    /* boilerplate: shutdown (omitted here) */
26
27    return 0;
28 }

```

Figure 6.2: RMARaceBench local buffer race example of the *miscellaneous* category: The call to *memcpy* defines an alias to the local buffer *buf* and the conflicting local memory access is performed on the alias. As discussed in Section 5.3.1, this leads to a false negative in RMA Sanitizer when the static filtering is activated since the access in line 20 will be falsely identified as irrelevant, missing the local buffer race.

test cases using MPI RMA, but they only cover simple argument errors and misplaced calls and do not include MPI RMA race tests. The test cases were used to evaluate the classification quality of different MPI verification tools. In an extended variant [42] of MPI-Corrbench, further test cases using MPI+OpenMP focusing on errors in the interaction of MPI and OpenMP have been added. Like MPI-Corrbench, the MPI Bugs Initiative (MBI) [57] also covers different MPI error classes and defines corresponding test cases. MBI uses a templating system to generate 1700 tests based on a few template scripts. MBI also contains local buffer races and remote races in MPI RMA, a total of 45 tests. However, it considers a smaller set of MPI RMA completion modes (fences, lock/unlock) than RMA RaceBench, which also covers flushes, PSCW, and synchronization via barriers or send/receive. Further, incorrect atomics and the interaction of MPI RMA with OpenMP are not considered.

The Run-Time Error Detection (RTED) suite [63] is another effort comprising 10000 manually written error test cases for MPI, OpenMP, and UPC programs. RTED also provides 86 MPI RMA tests with data races but only supports the MPI-2.0 standard. Thus, it only considers the separate memory model and misses the unified memory model, which is the predominant model in today's MPI RMA applications. Further, it does not

consider the new completion mechanisms (flushes, lock-all) introduced with MPI-3.0. Also, RTED does not provide correct test cases, so it cannot be used for classification quality evaluation.

The efforts of MPI-Corrbench, the MPI Bugs Initiative, and RMARaceBench have been merged into a collaborative project named MPI-BugBench [44] that I also contributed to. MPI-BugBench reuses the test code generation infrastructure of the MPI Bugs Initiative and additionally integrates the test cases of MPI-Corrbench and the MPI RMA race test cases of RMARaceBench to enhance the test suite’s coverage. Further, MPI-BugBench can generate different sets of test cases depending on the desired coverage level, ranging from 128 basic tests to 4 million generated test cases. An evaluation of the coverage of real-world usage patterns based on [41] shows that MPI-BugBench has a significantly higher coverage than MPI-Corrbench or the MPI Bugs Initiative. MPI-BugBench is open-source and publicly available<sup>3</sup>. Since MPI-BugBench only supports MPI, the following evaluation still uses RMARaceBench which includes SHMEM and GASPI test cases.

### 6.1.3 RMARaceBench Results

The test cases of RMARaceBench have been evaluated with RMASanitizer and other RMA race detectors to showcase and compare their detection accuracy. RMASanitizer was tested in two variants: The first variant, named *RMASanitizer*, is the default execution with the static analysis to filter out irrelevant local memory accesses, and the second variant, *RMASanitizer-NoOpt*, is the execution without static analysis, i.e., all local memory accesses are instrumented.

To compare RMASanitizer with other state-of-the-art RMA race detectors, the dynamic MPI RMA race detector of PARCOACH described in Section 5.1.4 and the static MPI RMA race detector for local buffer races of PARCOACH described in Section 5.1.7 were also tested with RMARaceBench. In the following, those tools will be named *PARCOACH-dynamic* and *PARCOACH-static*, respectively. The original RMARaceBench paper [91] tested MUST-RMA [90], which is the predecessor of RMASanitizer. The results of MUST-RMA are similar to those of RMASanitizer and are also included in the following evaluation. Other tools are not publicly available and could not be tested.

All outputs and results on the different test cases are available in the thesis artifact in Appendix A which also includes the test infrastructure for reproducibility.

#### Setup

RMARaceBench provides a test harness in a Docker environment for a reproducible evaluation. The tests were executed using a Debian 12 image with OpenMPI 4.1.4<sup>4</sup> for

---

<sup>3</sup><https://git-ce.rwth-aachen.de/hpc-public/mpi-bugbench>

<sup>4</sup><https://www.open-mpi.org>

MPI RMA, Sandia OpenSHMEM 1.5.1<sup>5</sup> for SHMEM, and GPI-2 1.5.1<sup>6</sup> for GASPI. The RMASanitizer execution uses the Clang 16 compiler, while the PARCOACH execution uses the Clang 15 compiler, as the tools require those compiler versions. There is no difference in the resulting behavior of the test cases between the two different compiler versions. For RMASanitizer and MUST-RMA, the tool versions provided in the thesis artifact are used. For PARCOACH, the tool version<sup>7</sup> recommended by its authors is used. RMASanitizer is executed with all test cases for MPI RMA, SHMEM, and GASPI, while MUST-RMA and PARCOACH only support MPI RMA and therefore are only executed with the MPI RMA test cases. As the tests are kept simple, most of the tests require the execution with only two or three processes. At maximum, four processes are required. The hybrid category tests always use two processes and up to two OpenMP threads per process.

## Metrics

The tool results on the test cases are interpreted as a binary classification. RMARaceBench classifies each run on a test case as follows [91]:

- True Positive (TP): The tool correctly reports a race in a test case *with a race*.
- True Negative (TN): The tool does not report a race in a *race-free* test case.
- False Positive (FP): The tool falsely reports a race in a *race-free* test case.
- False Negative (FN): The tool does not report a race in a test case *with a race*.
- Timeout (TO): The execution did not finish within 30 seconds.

To achieve a true positive (TP), a race report is only considered correct if the tool correctly reports the (two) affected source code lines. If the tool reports a race only for unrelated source code lines or variables, then this is considered a false negative (FN) because a tool report pinpointing the wrong location is not helpful at all. This interpretation of true positives (TP) has also been used in other research works [43, 44]. A timeout (TO) of 30 seconds is reasonable for all test cases because each test only requires a few milliseconds to finish without a tool.

Based on the number of TP, TN, FP, and FN results, RMARaceBench calculates the usual binary classification metrics for each tool: Precision  $P = \frac{TP}{TP+FP}$ , Recall  $R = \frac{TP}{TP+FN}$ , and Accuracy  $A = \frac{TP+TN}{TP+TN+FP+FN}$ . The fewer false positives (FP) a tool reports, the higher the precision  $P$ . False reports should be avoided, as they may distract the user from finding the actual bug or falsely tell the user to find a non-existent bug. The more races a tool can correctly detect in a given test set, the higher its recall  $R$ . The accuracy finally quantifies the overall detection quality of the tool.

<sup>5</sup><https://github.com/Sandia-OpenSHMEM/SOS>

<sup>6</sup><https://github.com/cc-hpc-itwm/GPI-2>

<sup>7</sup><https://gitlab.inria.fr/parcoach/parcoach/-/tree/Merge-non-adjacent+Flush>, commit hash 41584a46259e8178b3c811f97882c19b9103e08d

## 6 Classification Quality and Overhead Analysis of RMA Race Detection

Table 6.2: Classification quality results for the MPI RMA tests of RMA RaceBench for the different RMA race detectors.

| MPI RMA Results     |       |             |    |    |    |    |      |      |      |   |
|---------------------|-------|-------------|----|----|----|----|------|------|------|---|
| Tool                | Total | Tool Result |    |    |    |    | TO   | P    | R    | A |
|                     |       | TP          | TN | FP | FN | TO |      |      |      |   |
| PARCOACH-static     | 45*   | 17          | 10 | 8  | 10 | 0  | 0.68 | 0.63 | 0.60 |   |
| PARCOACH-dynamic    | 126   | 28          | 26 | 7  | 27 | 38 | 0.80 | 0.51 | 0.61 |   |
| MUST-RMA            | 126   | 51          | 52 | 1  | 22 | 0  | 0.98 | 0.70 | 0.82 |   |
| RMA Sanitizer       | 126   | 67          | 53 | 0  | 6  | 0  | 1.00 | 0.92 | 0.95 |   |
| RMA Sanitizer-NoOpt | 126   | 70          | 53 | 0  | 3  | 0  | 1.00 | 0.96 | 0.98 |   |

TP = True Positive, FP = False Positive, TN = True Negative, FN = False Negative, TO = Timeout

P = Precision, R = Recall, A = Accuracy

\*PARCOACH-static only detects local buffer races, so only those tests are considered for that tool.

Table 6.3: Classification quality metrics for the tools on the different test categories on MPI RMA in RMA RaceBench. The value “-” indicates that the metric is undefined since its denominator would be 0.

|                     | Conflict |      |      | Sync |      |      | Atomic |      |      | Hybrid |      |      | Miscellaneous |      |      |
|---------------------|----------|------|------|------|------|------|--------|------|------|--------|------|------|---------------|------|------|
|                     | P        | R    | A    | P    | R    | A    | P      | R    | A    | P      | R    | A    | P             | R    | A    |
| PARCOACH-static     | 1.00     | 0.42 | 0.53 | 0.62 | 0.83 | 0.67 | -      | -    | -    | 0.44   | 0.80 | 0.40 | 1.00          | 0.75 | 0.88 |
| PARCOACH-dynamic    | 0.93     | 0.54 | 0.67 | 0.71 | 0.50 | 0.63 | 0.33   | 0.33 | 0.20 | 1.00   | 1.00 | 1.00 | 1.00          | 0.33 | 0.62 |
| MUST-RMA            | 1.00     | 0.50 | 0.67 | 0.95 | 0.95 | 0.95 | -      | 0.00 | 0.40 | 1.00   | 0.83 | 0.91 | 1.00          | 1.00 | 1.00 |
| RMA Sanitizer       | 1.00     | 1.00 | 1.00 | 1.00 | 0.95 | 0.97 | 1.00   | 1.00 | 1.00 | 1.00   | 0.83 | 0.91 | 1.00          | 0.67 | 0.83 |
| RMA Sanitizer-NoOpt | 1.00     | 1.00 | 1.00 | 1.00 | 0.95 | 0.97 | 1.00   | 1.00 | 1.00 | 1.00   | 0.83 | 0.91 | 1.00          | 1.00 | 1.00 |

P = Precision, R = Recall, A = Accuracy

### Classification Results

Table 6.2 shows the results of the tools on the MPI RMA test cases. The results on SHMEM and GASPI will be discussed later, as those tests were only evaluated with RMA Sanitizer. As PARCOACH-static only supports the detection of local buffer races, it was only tested with a reduced set of 45 test cases. Detailed results for each test case are available in Appendix B and in the thesis artifact.

In summary, PARCOACH-static and PARCOACH-dynamic both have an accuracy  $A$  of roughly 0.60 with both false positives and negatives, leading to a relatively low precision  $P$  and recall  $R$ . The main reason is that both only understand a subset of the completion mechanisms in MPI RMA, as will be detailed later. MUST-RMA has a higher accuracy  $A$  of 0.82, with only one false positive (FP) but still a high number of false negatives (FN). RMA Sanitizer has the highest accuracy of 0.95 with no false positive (FP) and only 6 false negatives (FN) for MPI RMA. With static memory access filtering disabled in

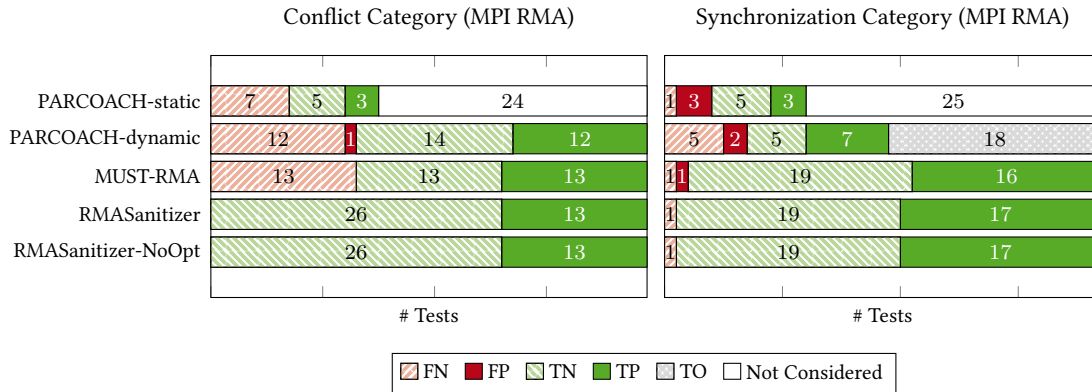


Figure 6.3: Detailed evaluation results for the conflict and synchronization category on the MPI RMA test cases for the different tools.

RMA Sanitizer-NoOpt, the accuracy  $A$  even gets up to 0.98. To compare the tools in detail, the metric scores achieved in the different categories for MPI RMA are depicted in Table 6.3. The results for each category are explained and discussed in detail below.

The *conflict* category iterates through the different combinations of all RMA communication calls. In this category, all tools show a very high precision  $P$ , i.e., if a race is reported by a tool, then there is indeed a race in the test. On the other hand, the recall  $R$  is low for PARCOACH-static, PARCOACH-dynamic, and MUST-RMA, because they only analyze the basic RMA calls *MPI\_Put*, *MPI\_Get*, and *MPI\_Accumulate*. The advanced MPI RMA atomics such as *MPI\_Get\_accumulate*, *MPI\_Fetch\_and\_op*, and *MPI\_Compare\_and\_swap* are not considered, so races with those calls are not detected. Those calls are only supported by RMA Sanitizer, achieving an accuracy  $A$  of 1 in this category. Figure 6.3 shows the absolute numbers of the evaluation results for the *conflict* category. PARCOACH-dynamic and MUST-RMA have a similar number of false negatives (FN), primarily related to non-supported RMA atomics. Further, PARCOACH-dynamic has one false positive (FP) for two concurrent *MPI\_Accumulate* calls targeted to the same memory location, as it does not understand the atomicity of RMA calls at all. Additionally, it has a false negative (FN) for a concurrent conflicting *MPI\_Get* and local store at the target process, hinting at a wrong conflict analysis of the tool for this combination.

The *synchronization* category iterates through different completion modes of the RMA models. In the case of MPI RMA, as shown in Figure 6.3, the tools deliver mixed results: PARCOACH-static has 3 false positives (FP) and 1 false negative (FN), resulting in an accuracy of 0.67. The 3 false positives (FP) are test cases that use local completion with *MPI\_Win\_lock/unlock*, *MPI\_Win\_flush\_local\_all*, and PSCW, which PARCOACH-static does not seem to consider in its analysis. On the other hand, the false negative (FN) test case uses request-based MPI RMA communication calls which PARCOACH-static also seems to ignore. PARCOACH-dynamic has several false positives (FP) and false negatives (FN) in this category, with an accuracy of 0.63. Since PARCOACH-dynamic assumes RMA programs with collective completion (*MPI\_Win\_fence*, or collective *MPI\_Win\_unlock\_all*),

```

1 int main(int argc, char** argv)
2 {
3     /* boilerplate: initialization (omitted here) */
4     MPI_Barrier(MPI_COMM_WORLD);
5
6     if (rank == 0) {
7         MPI_Win_lock(MPI_LOCK_EXCLUSIVE, 1, 0, win);
8         // remote write to win_base[0] at rank 1
9         MPI_Put(&value, 1, MPI_INT, target, 0, 1, MPI_INT, win); // CONFLICT
10        MPI_Win_unlock(1, win);
11        // lock own window (win2, rank 0)
12        MPI_Win_lock(MPI_LOCK_EXCLUSIVE, 0, 0, win2);
13        MPI_Win_unlock(0, win2);
14    } else {
15        sleep(1); // make it probable that rank 0 first locks win2
16        // lock window (win2, rank 0)
17        MPI_Win_lock(MPI_LOCK_EXCLUSIVE, 0, 0, win2);
18        // local read of win_base[0] at rank 1
19        printf("win_base[0] is %d\n", win_base[0]); // CONFLICT
20        MPI_Win_unlock(0, win2);
21    }
22
23    MPI_Barrier(MPI_COMM_WORLD);
24    /* boilerplate: shutdown (omitted here) */
25 }

```

Figure 6.4: RMARaceBench remote race example of the *synchronization* category: The `MPI_Put` in line 9 is in conflict with the local read in line 19. The synchronization established by the locking on `win2` between rank 0 and rank 1 would establish synchronization if `MPI_Win_lock` in line 17 was blocking. However, `MPI_Win_lock` is not required to block, so this is a remote race. As MUST-RMA and RMA Sanitizer assume that `MPI_Win_lock` blocks, they assume synchronization and do not detect a data race here.

only a small subset of the test cases is classified correctly. For 18 of the test cases, PARCOACH-dynamic even deadlocks without any result. The affected test cases use non-collective completion modes (locks, flushes, PSCW).

MUST-RMA and RMA Sanitizer show a very high accuracy in the *synchronization* category. Since they support all completion modes and the clock-based synchronization tracking can capture the individual synchronization between processes, both yield better results than PARCOACH-dynamic. MUST-RMA has one false positive (FP) for a test case containing conflicting RMA calls that are externally synchronized. The corresponding test case is similar to the example discussed in Figure 4.8. RMA Sanitizer understands externally synchronized RMA calls and avoids this false positive. The false negative (FN) is the same for both MUST-RMA and RMA Sanitizer: It is due to the overapproximated synchronization of `MPI_Win_lock` that has been discussed in Section 3.2.2. The corresponding example is shown in Figure 6.4. The rather complex code example shows that situations where the overapproximated modeling of `MPI_Win_lock` matters are highly artificial and probably irrelevant for real-world applications.

The *atomic* category considers remote races resulting from violated atomicity conditions, e.g., incompatible data types or offsets. RMA Sanitizer is the only tool that supports detecting such races, so it achieves an accuracy of 1, as indicated in Table 6.3. The atomicity

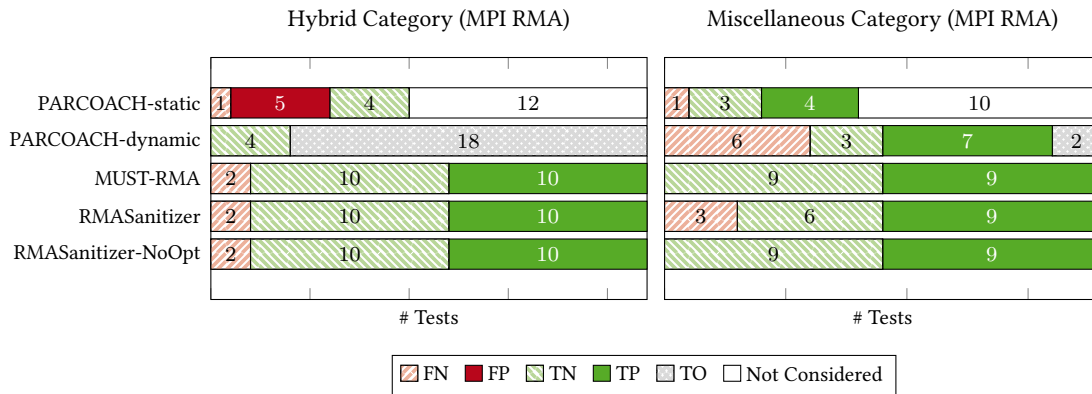


Figure 6.5: Detailed evaluation results for the hybrid and miscellaneous category on the MPI RMA test cases for the different tools.

check of RMA Sanitizer described in Section 5.3.5 ensures the correct classification of all test cases.

The *hybrid* category combines the RMA models with OpenMP to verify if an RMA race detector can understand the interaction between both. PARCOACH-static scans through the code without considering OpenMP constructs, which should, in theory, lead to many false negatives (FN) because when the code is interpreted as sequential code, then there is no race. However, PARCOACH-static surprisingly has many false positives (FP), as shown in Figure 6.5. All programs are compiled with OpenMP support, i.e., OpenMP regions are outlined in separate functions. This confuses PARCOACH-static, hinting at a problem in the case of races across function calls due to a missing interprocedural analysis. The test cases of the *hybrid* category were designed such that individual synchronization between the threads within a process and across processes has to be understood by the race detector for correct classification. PARCOACH-dynamic deadlocks in nearly all test cases as it does not support fine-grained synchronization in RMA. Although MUST-RMA and RMA Sanitizer also do not support hybrid parallelism with OpenMP, they can classify nearly all test cases correctly. Both rely on ThreadSanitizer for the race detection which comes with Archer [3] that tracks OpenMP threading and synchronization. Consequently, the annotation of RMA concurrent regions to ThreadSanitizer is correctly interpreted in the context of OpenMP threading, resulting in many correctly classified tests. However, as discussed in Section 3.6, the vector clock exchange of RMA Sanitizer overapproximates synchronization between individual threads of two processes. This leads to the 2 false negatives (FN) in the detection of remote races.

Lastly, the *miscellaneous* category contains the test cases that were designed to challenge the static analysis of RMA Sanitizer. It is the only category where the results of RMA Sanitizer and RMA Sanitizer-NoOpt differ. As depicted in Figure 6.5, RMA Sanitizer has three false negatives (FN), where two of them are due to aliasing via the external function *memcpy*. The corresponding code is similar to the example in Figure 6.2. The third one is due to the use of function pointers which cannot be statically resolved. All other

Table 6.4: Classification quality results for the GASPI and SHMEM tests of RMA RaceBench for RMA Sanitizer.

| SHMEM Results       |       |             |    |    |    |    |      |      |      |
|---------------------|-------|-------------|----|----|----|----|------|------|------|
| Tool                | Total | Tool Result |    |    |    |    | P    | R    | A    |
|                     |       | TP          | TN | FP | FN | TO |      |      |      |
| RMA Sanitizer       | 122   | 65          | 48 | 1  | 8  | 0  | 0.98 | 0.89 | 0.93 |
| RMA Sanitizer-NoOpt | 122   | 69          | 48 | 1  | 4  | 0  | 0.99 | 0.95 | 0.96 |

| GASPI Results       |       |             |    |    |    |    |      |      |      |
|---------------------|-------|-------------|----|----|----|----|------|------|------|
| Tool                | Total | Tool Result |    |    |    |    | P    | R    | A    |
|                     |       | TP          | TN | FP | FN | TO |      |      |      |
| RMA Sanitizer       | 100   | 57          | 38 | 0  | 5  | 0  | 1.00 | 0.92 | 0.95 |
| RMA Sanitizer-NoOpt | 100   | 59          | 38 | 0  | 3  | 0  | 1.00 | 0.95 | 0.97 |

TP = True Positive, FP = False Positive, TN = True Negative, FN = False Negative, TO = Timeout  
P = Precision, R = Recall, A = Accuracy

aliasing and interprocedural test cases are classified correctly. PARCOACH-dynamic also uses static analysis to filter out irrelevant memory accesses, as discussed in Section 5.1.4. However, its analysis does not support interprocedural analysis, leading to a higher number of non-instrumented relevant local memory accesses. The test results confirm this: When a relevant local memory access is made to an RMA buffer but isolated in a separate function, it is not instrumented, resulting in a false negative (FN). Like with RMA Sanitizer, some aliasing test cases are also not correctly classified in PARCOACH-dynamic. PARCOACH-static characterizes the test cases correctly. There is only one false negative (FN) in a test case with deeply nested function calls.

Table 6.4 shows the results of RMA Sanitizer on the SHMEM and GASPI test cases. The results are overall similar to that of MPI RMA. The accuracy  $A$  of RMA Sanitizer ranges from 0.93 to 0.97, depending on whether the static analysis is activated. The results confirm that RMA Sanitizer also captures the completion and synchronization mechanisms of those RMA models correctly. The false negatives (FN) are mainly equivalent to the test cases of the MPI RMA variant, i.e., there are some false negatives in the *hybrid* category, and some false negatives in the *miscellaneous* category due to the static analysis. There is an additional false negative (FN) in SHMEM related to the *atomic* category: RMA Sanitizer currently does not correctly detect atomicity violations when using different SHMEM contexts in different atomicity domains. This is a corner case whose detection would require significant additional implementation effort by also encoding the underlying atomicity domain of SHMEM contexts, and has been omitted. Further, there is one false positive (FP) in SHMEM when using `shmem_put_signal` in combination with polling `shmem_signal_fetch`, as shown in Figure 6.6. RMA Sanitizer currently does not capture this specific polling-based synchronization, leading to a falsely detected race. The main reason is that supporting `shmem_signal_fetch` in SHMEM would also require larger

```

1 int main(int argc, char** argv)
2 {
3     /* boilerplate: initialization (omitted here) */
4     static int remote = 0;
5     static uint64_t signal = 0;
6
7     shmem_barrier_all();
8
9     if (my_pe == 0) {
10        // polling on signal
11        while (shmem_signal_fetch(&signal) != 1) {}
12        printf("Remote: %d\n", remote);
13    }
14
15    if (my_pe == 1) {
16        remote = 42;
17        // write to remote at process 0 and signal completion
18        shmem_int_put_signal(&remote, &my_pe, 1, &signal, 1, SHMEM_SIGNAL_ADD, 0);
19    }
20    shmem_barrier_all();
21
22    /* boilerplate: shutdown (omitted here) */
23 }

```

Figure 6.6: RMARaceBench SHMEM example with no race: The *shmem\_int\_put\_signal* call at process 1 in line 18 writes to the variable *remote*. Process 0 waits for the signal to be set by process 1 in line 11. The *printf* access to *remote* in line 12 is thus not concurrent. RMA Sanitizer currently does not consider this particular signal synchronization in SHMEM, so it assumes concurrent accesses, leading to a falsely reported race.

adaptations in the implementation. Other notify-wait synchronization, such as the blocking *shmem\_wait\_until* instead of polling via *shmem\_signal\_fetch*, are detected and correctly classified. In general, any synchronization mechanism not captured or ignored by RMA Sanitizer potentially leads to false positives (FP). In those cases, user annotations, as discussed in Section 3.4.6, can be added to the code to avoid them.

Another corner case currently not covered in all variants by RMA Sanitizer is implicit local completion with notify-wait synchronization. As already outlined in Section 4.2.2, the concurrent region of local buffer accesses in the case of notify-wait is sometimes difficult to determine when it is ensured implicitly as it may require propagating additional consistency information across processes. Figure 6.7 shows an example where the local concurrent region of a local buffer read is not ended explicitly through a local completion call, i.e., there is no local completion call used by process 0. To avoid a false positive in the given example, RMA Sanitizer assumes that target wait events such as *shmem\_wait\_until* ensure local completion of previously issued local buffer reads associated with corresponding remote writes. This is, however, an overapproximation of the completion behavior: In Figure 6.7, if the *shmem\_wait\_until* call in line 17 is removed, then there is indeed a local buffer race in the code which is not detected by RMA Sanitizer, leading to a false negative (FN). Again, this limitation only applies to local buffer read accesses of RMA operations completed via notify-wait completion, and even then, only in certain scenarios. In future extensions, RMA Sanitizer might be extended to consider the implicit completion of local buffer reads more precisely. This would require propagating

```

1 int main(int argc, char** argv)
2 {
3     /* boilerplate: initialization (omitted here) */
4     int localbuf = 1; static int remote = 0; static uint64_t signal = 0;
5
6     shmem_barrier_all();
7
8     if (my_pe == 0) {
9         localbuf = 42;
10        // send data with signal (ping)
11        shmem_int_put_signal_nbi(&remote, &localbuf, 1, &signal, 1, SHMEM_SIGNAL_SET, 1);
12        shmem_uint64_wait_until(&signal, SHMEM_CMP_EQ, 1); // wait for pong from PE 1
13        localbuf = 1337;
14    }
15
16    if (my_pe == 1) {
17        shmem_uint64_wait_until(&signal, SHMEM_CMP_EQ, 1); // wait for ping from PE 0
18        // send data with signal (pong)
19        shmem_int_put_signal(&remote, &localbuf, 1, &signal, 1, SHMEM_SIGNAL_SET, 0);
20    }
21
22    shmem_barrier_all();
23
24    /* boilerplate: shutdown (omitted here) */
25 }

```

Figure 6.7: RMARaceBench SHMEM example achieving local completion through notifications: The `shmem_int_put_signal_nbi` call at process 0 in line 11 is completed implicitly through notifications. Process 1 first waits for the signal to arrive in line 17. Then, it sends back a signal in line 19 which is finally received in line 12. This also means that the conflicting store to `localbuf` in line 13 is not concurrent to the local buffer access of `shmem_int_put_signal_nbi`: The access is guaranteed to be (remotely) completed at process 1 in line 17 which then propagates to the `wait-until` call in line 12.

the information of an RMA operation's remote completion back to the origin process when *any* synchronization with the origin process is performed, i.e., when the origin has to wait for the target. Then, the origin can deduce whether local completion is ensured. This extension would require larger adaptations of the implementation and also the concurrent region model. In addition to the example for SHMEM in Figure 6.7, RMARaceBench also comes with a corresponding example in GASPI that shows the same behavior. For MPI RMA, there is no notification mechanism, so this limitation leading to false negatives does not apply.

### 6.1.4 Discussion

The results show that RMA Sanitizer has a high detection accuracy on RMA RaceBench in all test categories. The clock-based approach to find concurrent regions of RMA operations effectively detects RMA races while avoiding falsely reported data races. Even in the case of hybrid programs, RMA Sanitizer classifies most of the tests correctly since its race detection partly relies on ThreadSanitizer which has with Archer [3] mature tool support for OpenMP synchronization.

RMASanitizer has a higher detection accuracy in RMA RaceBench than PARCOACH-dynamic that only supports RMA programs with collective synchronization. The same is true for PARCOACH-static, which also only supports a limited set of completion mechanisms. Compared to MUST-RMA, RMASanitizer improves the support of atomic operations and some further corner cases with external synchronization.

The static analysis of RMASanitizer only has a negligible impact on the detection accuracy in RMA RaceBench. The results confirm that the static analysis might lead to false negatives in specific scenarios but never to false positives. If the highest detection accuracy is required, then the static analysis should be deactivated.

RMASanitizer has a high coverage on the completion modes in MPI RMA, SHMEM, and GASPI. Still, there might be some corner cases, such as the discussed signal-fetch mechanism in SHMEM, that are currently not fully supported. For any non-supported synchronization concept, user annotations can still be used to avoid false reports of RMASanitizer. Further, some rare scenarios of implicit local completion are difficult to track completely and therefore overapproximated by RMASanitizer leading to missed local buffer races. Future extensions may also include the detection of such situations.

The tests of RMA RaceBench are deterministic in the sense that the RMA data race is always observable in any execution run, i.e., the affected source code lines are always executed. As with any (dynamic) race detector relying on the happened-before relation, RMASanitizer cannot detect a data race when an alternative execution path is taken such that the data race is not visible. From this perspective, static race detection tools such as PARCOACH-static achieve better coverage because they do not rely on the program execution order. However, approaches such as Nasty-MPI [48] enforcing worst-case execution in MPI RMA or DAMPI [108] using replay techniques may help to explore all execution paths in a non-deterministic program execution combined with a dynamic tool such as RMASanitizer.

## 6.2 Overhead Analysis

A correctness tool is only useful in practice if it is applicable to real-world workloads. The relevant metric for on-the-fly analysis tools is the introduced runtime overhead. RMASanitizer has been tested on different RMA proxy applications to showcase its applicability. The following sections discuss the chosen proxy applications and the overhead results of RMASanitizer in detail. The applications cover all three programming models, MPI RMA, SHMEM, and GASPI, and different communication and completion variants.

Table 6.5: Set of RMA proxy applications that were tested with RMA Sanitizer.

| Application | Description                | Model          | RMA Variant                |
|-------------|----------------------------|----------------|----------------------------|
| PRK Stencil | 5-point stencil on 2D grid | MPI RMA, SHMEM | put + collective flush     |
| NPB BT*     | Block tridiagonal solver   | MPI RMA, SHMEM | get/put + collective flush |
| miniMD*     | Molecular dynamics         | MPI RMA        | get + remote flush         |
| LULESH*     | Hydrodynamics stencil      | MPI RMA        | get + collective flush     |
| CFD-Proxy   | CFD on unstructured mesh   | GASPI          | put + notifications        |
| miniVite    | Graph communities          | MPI RMA        | put + remote flush         |

\* The original application does not use RMA but has been ported to the listed RMA programming model(s).

## 6.2.1 Applications

RMA programming models such as MPI RMA are less widespread than the traditional distributed-memory programming via MPI point-to-point or collectives, as pointed out by Laguna et al. [53]. Therefore, the number of available applications to test RMA Sanitizer is limited. To pinpoint RMA Sanitizer’s strengths and weaknesses, a set of proxy applications using different RMA communication patterns was chosen. Some are well-known proxy applications that have been ported to use RMA communication, while others were originally designed using RMA. Table 6.5 lists the proxy applications tested with RMA Sanitizer.

The Parallel Research Kernels (PRK) [105] are a collection<sup>8</sup> of simple kernels parallelized using different parallel programming models. PRK Stencil provides a C implementation of a 5-point stencil on a 2D grid using MPI RMA and SHMEM. It uses different variants of RMA for the halo exchange with RMA puts combined with collective flushes. The MPI RMA variant uses *MPI\_Put* operations together with *MPI\_Win\_fence* and the SHMEM variant uses *shmem\_put* in combination with *shmem\_barrier\_all*. The original SHMEM code in PRK relies on *shmem\_wait\_until* and *shmem\_put*, which would result in undefined behavior according to the newest OpenSHMEM standard [19], so it has been replaced for this evaluation with the aforementioned pattern instead.

The NAS parallel benchmarks<sup>9</sup> provide simple programs to evaluate the performance of supercomputers and are written in Fortran. The block tri-diagonal solver (BT) of NPB has been ported to MPI RMA by replacing *MPI\_Isend* and *MPI\_Irecv* calls with *MPI\_Put* and *MPI\_Win\_fence* for communication. The Oak Ridge OpenSHMEM Benchmarks<sup>10</sup> contain a SHMEM port of NPB BT using *shmem\_get* operations and *shmem\_barrier\_all*.

miniMD<sup>11</sup> is a C++ proxy application for molecular dynamics codes. It is a simplified variant of the algorithms provided in LAMMPS<sup>12</sup>. In miniMD, each process works on a

<sup>8</sup><https://github.com/ParRes/Kernels>

<sup>9</sup><https://www.nas.nasa.gov/software/npb.html>

<sup>10</sup><https://github.com/ornl-languages/osb>

<sup>11</sup><https://github.com/Mantevo/miniMD>

<sup>12</sup><http://lammps.sandia.gov>

subset of a simulation box and exchanges relevant data with neighbors using *MPI\_Sendrecv* calls. The application has been ported to MPI RMA by exchanging those *MPI\_Sendrecv* calls with *MPI\_Get* operations and using *MPI\_Win\_lock\_all/unlock\_all* combined with *MPI\_Barrier* for completion and synchronization.

LULESH<sup>13</sup> is a C++ application solving a hydrodynamics problem using stencil operations. It uses *MPI\_Isend* and *MPI\_Irecv* for the halo communication. The RMA port of this code replaces the *MPI\_Irecv* calls with *MPI\_Get* operations and uses *MPI\_Win\_fence* for completion and synchronization.

CFD-Proxy<sup>14</sup> is a computational fluid dynamics proxy application written in C and using GASPI. It implements a Green-Gauss gradient calculation kernel and provides different variants of RMA exchange patterns for the halo exchange. The variant chosen as a representative in the overhead evaluation uses *gaspi\_write\_notify* and *gaspi\_notify\_waitsome* to implement the halo exchange.

miniVite [33] is a C++ proxy application<sup>15</sup> for graph community detection using the Louvain algorithm parallelized with MPI. It requires exchanging vertex-community information between processes and supports different communication variants such as MPI collectives and MPI RMA. For the evaluation, the MPI RMA communication variant has been chosen. It uses *MPI\_Put* operations combined with *MPI\_Win\_lock\_all/unlock\_all* and *MPI\_Barrier* for completion and synchronization.

All codes are provided as part of the thesis artifact. The porting of miniMD and LULESH to MPI RMA has been done by Cornelius Pätzold as a student worker under my supervision. The porting of NPB BT to MPI RMA has been done by Markus Geimer and Marc-André Hermanns and revised by Cornelius Pätzold to work with the latest NPB build infrastructure.

## 6.2.2 Setup

The overhead experiments were performed on CLAX-2023 [18] consisting of 630 nodes connected via InfiniBand. Each node has 96 cores (two Intel Sapphire Rapids 8468) with SMT disabled and 256 GB of main memory available. The benchmarks were compiled with Clang/Classic-Flang 16 on the C/C++ and Fortran applications. The applications were executed with OpenMPI 4.1.6<sup>16</sup> for MPI RMA, Sandia OpenSHMEM 1.5.2<sup>17</sup> for SHMEM, and GPI-2 1.5.1<sup>18</sup> for GASPI.

<sup>13</sup><https://github.com/LLNL/LULESH>

<sup>14</sup><https://github.com/PGAS-community-benchmarks/CFD-Proxy>

<sup>15</sup><https://github.com/ECP-ExaGraph/miniVite>

<sup>16</sup><https://www.open-mpi.org>

<sup>17</sup><https://github.com/Sandia-OpenSHMEM/SOS>

<sup>18</sup><https://github.com/cc-hpc-itwm/GPI-2>

All previously mentioned applications have been compiled and executed with and without RMASanitizer to evaluate its runtime overhead. For the baseline measurement, the total execution time  $T_{Baseline}$  of the application is measured without any attached tool. For the tool measurement, the total execution time  $T_{RMASanitizer}$  of the application with the runtime analysis of RMASanitizer is measured. To additionally measure the impact of the static analysis filtering on the runtime analysis in RMASanitizer, all applications were executed with ( $T_{RMASanitizer}$ ) and without ( $T_{RMASanitizer-NoOpt}$ ) it. All time measurements show the average of five runs. The slowdown  $S = \frac{T_{RMASanitizer}}{T_{Baseline}}$  of RMASanitizer is computed and also plotted in the graphs. In summary, the measurement  $T_{Baseline}$  refers to the measurement without tool, the measurement  $T_{RMASanitizer}$  refers to the execution time when the static analysis filter was applied, and the measurement  $T_{RMASanitizer-NoOpt}$  is the same as  $T_{RMASanitizer}$  but with the static analysis filter disabled. The introduced compile time overhead of the static analysis itself has been measured in [92] to be 1.5x to 3x which makes the static analysis applicable also to larger code bases. The following evaluation focuses on the overhead of the resulting runtime analysis of RMASanitizer.

For a detailed breakdown analysis of the major tool components, (1) the consistency and synchronization tracking, i.e., the MUST part of RMASanitizer, and (2) the race detection in ThreadSanitizer were executed isolated in separate runs to measure their overhead individually, i.e., the following execution times were measured in separate runs:

- (1)  $T_{Baseline}$ , time without any tool (baseline),
- (2)  $T_{ThreadSanitizer}$ , time with ThreadSanitizer only,
- (3)  $T_{MUST}$ , time with the MUST part of RMASanitizer only, i.e., without ThreadSanitizer,
- (4)  $T_{RMASanitizer}$ , time with full RMASanitizer toolchain.

Those measurements are used later in the detailed breakdown analysis to pinpoint the root cause of the measured overheads by computing the following respective portions

- (1)  $R_{Application} = \frac{T_{Baseline}}{T_{RMASanitizer}}$ , application portion,
- (2)  $R_{ThreadSanitizer} = \frac{T_{ThreadSanitizer} - T_{Baseline}}{T_{RMASanitizer}}$ , ThreadSanitizer portion,
- (3)  $R_{MUST} = \frac{T_{MUST} - T_{Baseline}}{T_{RMASanitizer}}$ , RMASanitizer portion (MUST without TSan)
- (4)  $R_{Interaction} = 1 - R_{MUST} - R_{ThreadSanitizer} - R_{Application}$ , interaction of ThreadSanitizer and MUST component of RMASanitizer.

The applications were executed with different numbers of application processes, ranging from 32 to 768 processes, depending on the application's requirements. The concrete input parameters and the scalability setting (weak or strong) are annotated to the graphs.

As described in Chapter 5, RMASanitizer requires an additional tool thread for each application process. Therefore, the total number of 96 physical cores per node is split into two halves: The first half of the 48 cores is used for the application itself, and the second half is used for the tool threads. A spread binding is chosen to pin the application's cores

to even core numbers and the corresponding tool threads to odd core numbers. The baseline run only utilizes 48 of the total 96 physical cores per node, while the tool run uses the spare 48 cores for the tool threads. The same scheme has also been used in the vector clock exchange evaluation in Section 3.5.

Similar overhead measurements were done in my previous publication on RMA Sanitizer [93]. In addition to [93], this thesis adds a detailed breakdown analysis to pinpoint the root cause for the overheads. Further, the comparison to the results of PARCOACH in Section 6.2.4 is another addition compared to [93].

### 6.2.3 Overhead Results

This section gives an overview of the overhead results of RMA Sanitizer for the different tested applications by looking at the different slowdown factors and inspecting the responsible tool components in an overhead breakdown.

#### Weak Scaling Benchmarks

Figure 6.8 shows the results of RMA Sanitizer with PRK Stencil, LULESH, and miniMD, which all use a weak scaling setup. For PRK Stencil, the baseline execution requires around 100 s. Running RMA Sanitizer on PRK Stencil in the MPI RMA and the SHMEM variant has a very small slowdown of around 1.05x with 48 processes going up to a slowdown of 1.19x with 768 processes, i.e., the total execution time rises from 100 s to 119 s. A larger number of processes leads to larger vector clocks and a higher number of processes participating in the completion and synchronization in *MPI\_Win\_fence* and *shmem\_barrier\_all*, expectedly leading to a slightly higher slowdown. PRK Stencil performs only roughly  $10^3$  RMA operations per second, which is relatively low compared to the other tested applications.

Without the static analysis, depicted with *RMA Sanitizer-NoOpt*, the execution time rises from 119s to roughly 1000s, resulting in a slowdown of 10x. This shows that the static analysis is very effective in PRK Stencil: It can filter out all local memory accesses within the main stencil computation loop which is the hotspot of the application, so only the local memory accesses of the halo exchange are instrumented. Considering that PRK Stencil is a memory-bound code, reducing the number of local memory accesses in the main loop leads to a strong overhead reduction.

For LULESH and miniMD, the slowdown of RMA Sanitizer is higher than for PRK Stencil. As Figure 6.8 shows, the static analysis still effectively reduces the slowdown of RMA Sanitizer. LULESH and miniMD have a significantly more complex code structure than PRK Stencil (in terms of aliasing), making it more challenging to find irrelevant memory accesses for the static analysis. For LULESH, the slowdown of RMA Sanitizer ranges from 7x with 64 processes to 9x with 729 processes, and for miniMD, it ranges

## 6 Classification Quality and Overhead Analysis of RMA Race Detection

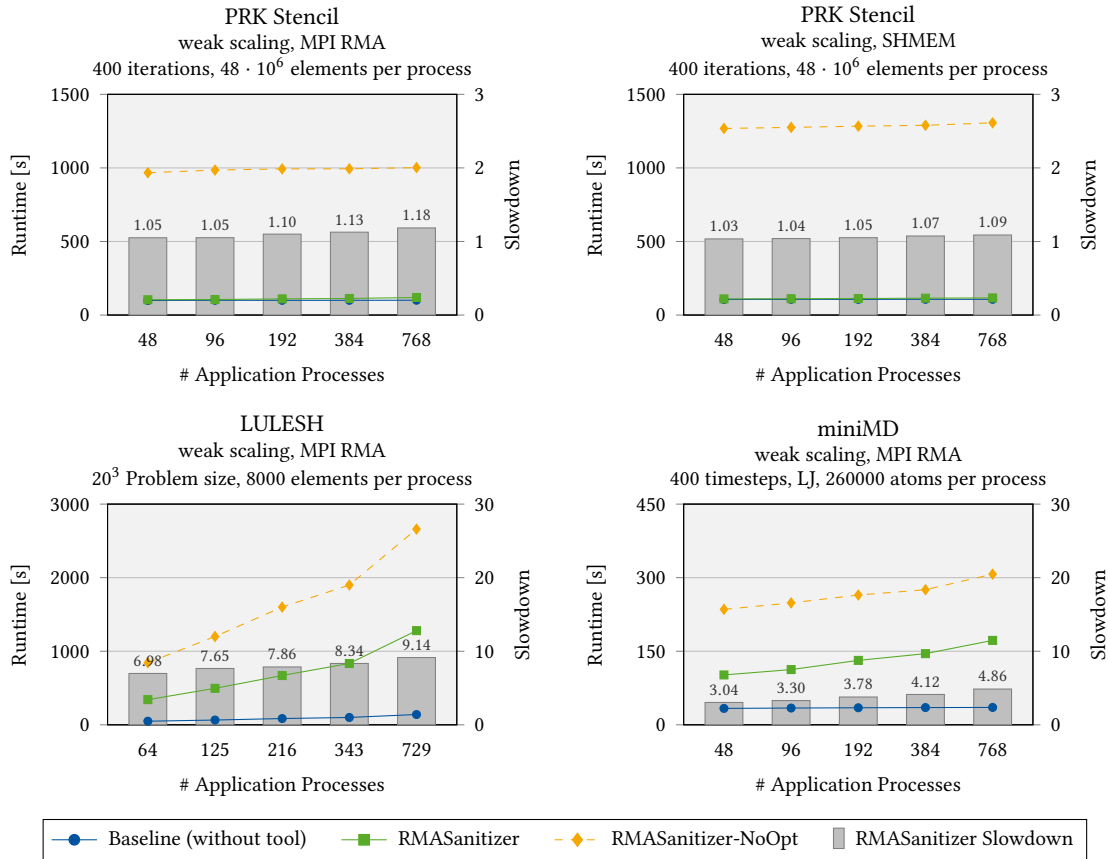


Figure 6.8: Overhead results of RMA Sanitizer on PRK Stencil, LULESH (ported), and miniMD (ported). The input parameters of each benchmark are annotated to the graphs. The left y-axis corresponds to the line plots showing the absolute runtime and the right y-axis corresponds to the bar plot showing the slowdown  $S = \frac{T_{RMA\text{Sanitizer}}}{T_{Baseline}}$  of RMA Sanitizer relative to the baseline. The measurement without static analysis,  $T_{RMA\text{Sanitizer-NoOpt}}$ , is also included for reference.

from 3x with 48 processes to 5x with 768 processes. The slowdown of RMA Sanitizer is expected to increase with the number of processes due to larger vector clocks and a larger number of processes that communicate through RMA Sanitizer’s infrastructure.

To confirm the previous statements, the overhead breakdown of the ThreadSanitizer portion and the MUST portion is shown in Figure 6.9 for PRK Stencil and LULESH. The interaction portion mainly represents the additional time due to the ThreadSanitizer annotations from RMA Sanitizer, i.e., the concurrent region annotations of RMA operations. For PRK Stencil in the *RMA Sanitizer-NoOpt* execution, ThreadSanitizer’s race analysis clearly dominates the overhead: Since *all* local memory accesses are instrumented and analyzed with ThreadSanitizer, it is a major overhead contributor. In the *RMA Sanitizer* execution, the ThreadSanitizer overhead vanishes as the static analysis eliminates the instrumentation of nearly all local memory accesses. Instead, the relative overhead of the MUST component increases slightly with larger process numbers.

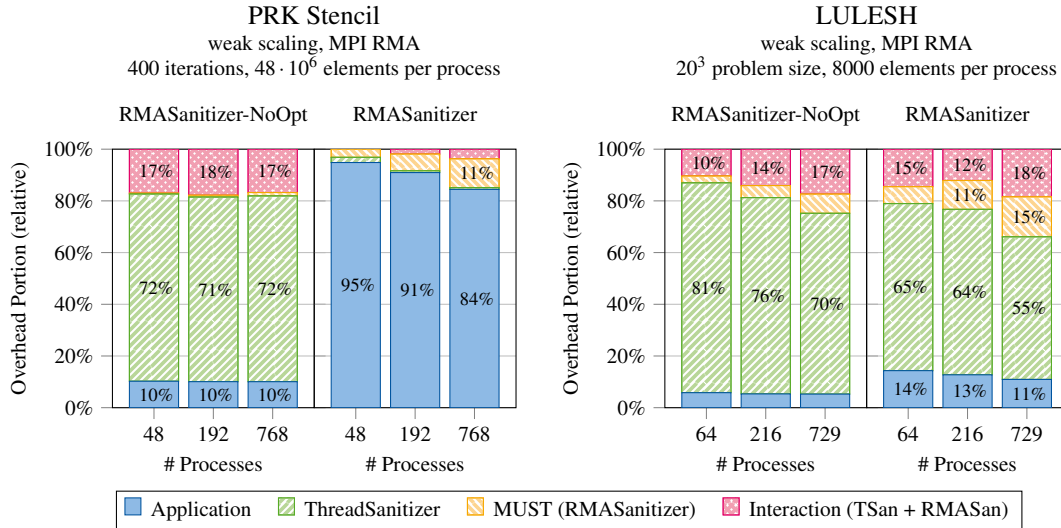


Figure 6.9: Overhead breakdown of the PRK Stencil (MPI RMA) and LULESH run. The bars show the percentage of RMASanitizer-NoOpt and RMASanitizer runs attributed to the different components. The resulting portions are (1)  $R_{Application}$  (2)  $R_{ThreadSanitizer}$ , (3)  $R_{MUST}$ , and (4)  $R_{Interaction}$ .

For LULESH, the static analysis is less effective, as shown in Figure 6.9, so the ThreadSanitizer overhead remains higher even with the static analysis. The relative overhead of the MUST component is expected to increase with the number of processes. The overhead breakdown shows that the ThreadSanitizer overhead remains the same for larger process numbers while the MUST overhead gets larger. This is also expected, as ThreadSanitizer itself only performs the data race analysis locally in each process. The MUST component has to track the synchronization and consistency between processes, which gets more complex with a higher number of processes. For example, the vector clock exchange in collective operations takes longer as more processes exchange their values.

### Strong Scaling Benchmarks

Figure 6.10 shows the results of RMASanitizer with NPB BT, CFD-Proxy, and miniVite which all use a strong scaling setup. For NPB-BT in the MPI RMA variant, the slowdown with RMASanitizer rises from 5x with 49 processes to 27x with 729 processes. The main reason for that is the very high amount of RMA operations it performs per second when scaling to a larger number of processes. With 49 processes, the application performs roughly  $10^3$  RMA operations per second, and with 729 processes, it raises to roughly  $10^6$  RMA operations per second. This is mainly due to the strong scaling setup, which reduces the amount of computation per process while keeping the frequency of RMA operations per process constant. A similar slowdown increase can be observed for NPB BT in the SHMEM variant. The static analysis is also less effective for this benchmark, as NPB BT uses many global variables, leading to many aliases.

## 6 Classification Quality and Overhead Analysis of RMA Race Detection

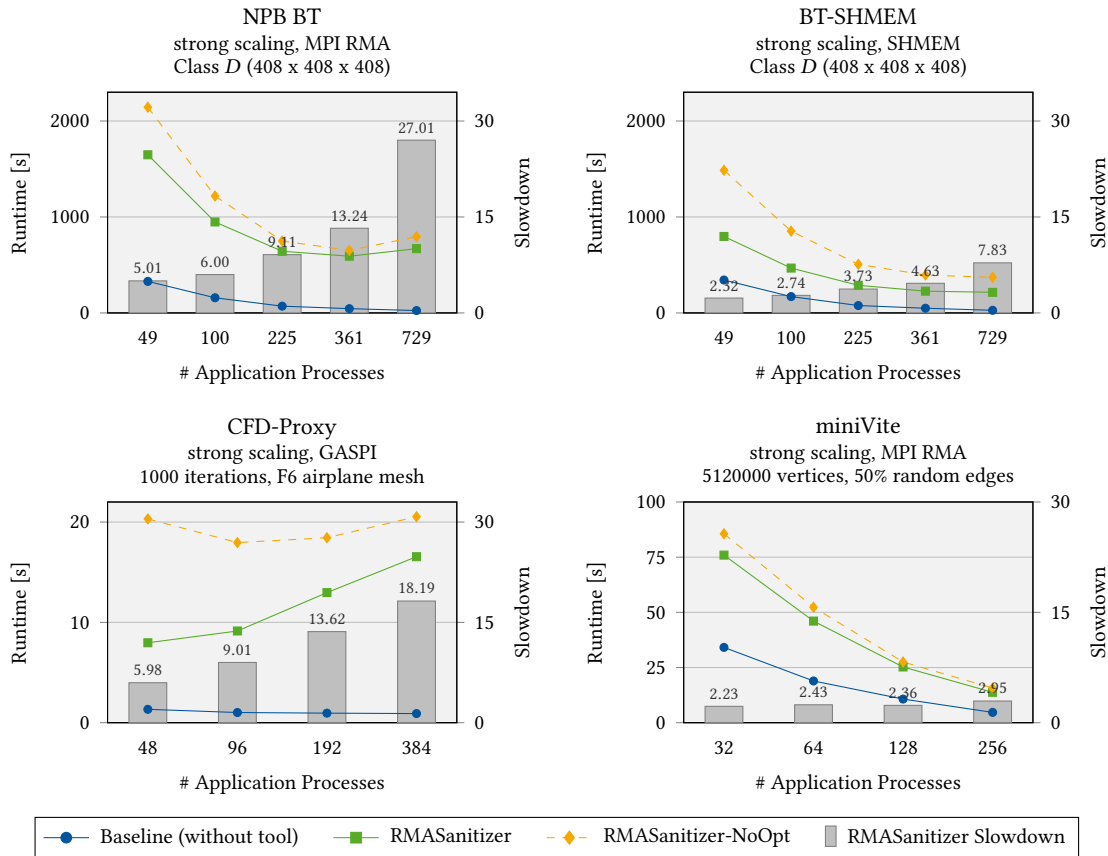


Figure 6.10: Overhead results of RMASanitizer on NPB BT (ported), CFD-Proxy, and miniVite. The input parameters of each benchmark are annotated to the graphs. The left y-axis corresponds to the line plots showing the absolute runtime and the right y-axis corresponds to the bar plot showing the slowdown  $S = \frac{T_{\text{RMASanitizer}}}{T_{\text{Baseline}}}$  of RMASanitizer relative to the baseline. The measurement without static analysis,  $T_{\text{RMASanitizer-NoOpt}}$ , is also included for reference.

For CFD-Proxy, only results with up to 384 processes are reported, as the input mesh does not support larger process numbers. Similar to NPB BT, the overhead significantly increases from 6x with 48 processes to 18x with 384 processes. Again, this results from strong scaling increasing the number of RMA operations per second from  $10^3$  with 48 processes to  $10^6$  with 384 processes. Further, the input mesh is very small, so only a few hundred mesh points are assigned to each process when running with 384 processes. Thus, communication dominates the already short execution time of a few seconds.

For miniVite, a randomly generated graph with similar parameters as in the experiment of the PARCOACH authors [107] is used as input. Only results with up to 256 processes are reported because miniVite crashes with a segmentation fault when using a larger number of processes. The overhead of RMASanitizer increases only slightly with the number of processes and is generally relatively low. This application performs a high number of *MPI\_Irecv* and *MPI\_Isend*, increasing quadratically with the number of processes, leading

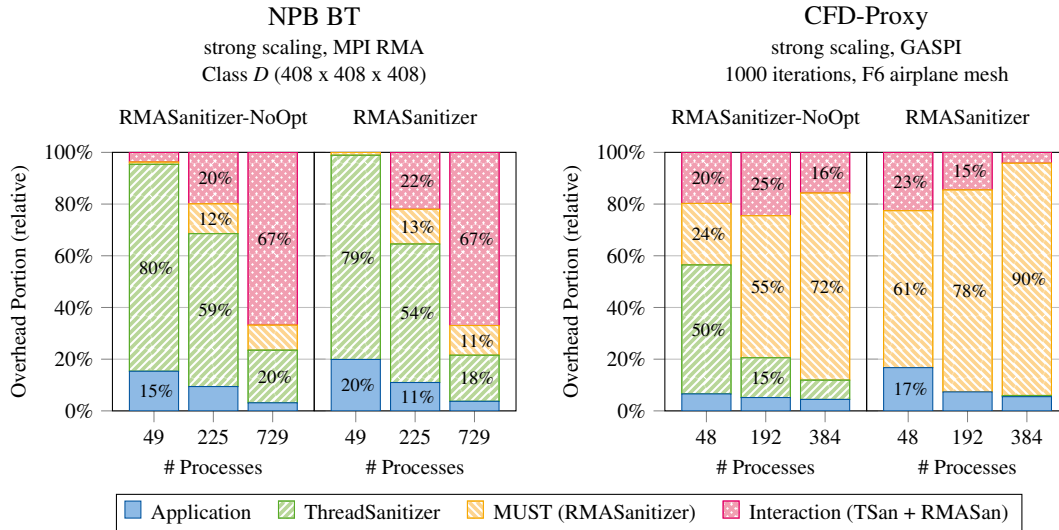


Figure 6.11: Overhead breakdown of NPB BT (MPI RMA) and CFD-Proxy. The bars show the percentage of RMA Sanitizer-NoOpt and RMA Sanitizer runs attributed to the different components. The resulting portions are (1)  $R_{Application}$  (2)  $R_{ThreadSanitizer}$ , (3)  $R_{MUST}$ , and (4)  $R_{Interaction}$ .

to a slightly higher pressure on the vector clock exchange analysis. The number of RMA operations is in the order of  $10^5$  RMA operations per second for 256 processes, i.e., lower than for the other strong scaling benchmarks.

The detailed overhead breakdown of NPB BT and CFD-Proxy in Figure 6.11 confirms that with an increasing number of processes, the overhead of the MUST component and its interaction with ThreadSanitizer dominates the overhead. The large number of RMA operations that must all be annotated from RMA Sanitizer to ThreadSanitizer leads to a high portion of interaction between the two tool components when running NPB BT. For CFD-Proxy, the interaction between RMA Sanitizer and ThreadSanitizer is also high, but the main overhead for larger process numbers is within the MUST component. The main reason for that is the vector clock exchange, which is very expensive in CFD-Proxy: After each iteration, several notifications for synchronization are individually sent between all the neighbors, leading to a massive amount of notifications that the analysis infrastructure must handle. For NPB BT, simple collective *MPI\_Win\_fence* (or *shmem\_barrier\_all*) calls are used, which have a significantly lower overhead in the vector clock analysis.

## Memory Overhead

The memory consumption of the different executions has been measured by retrieving the maximum resident set size (*max\_rss*) at the end of the execution to determine the peak memory consumption for each process. The per-process values were then summed up to retrieve the total memory consumption of the execution. The measured memory

overhead of RMA Sanitizer compared to the execution without tool is between 1.5x and 3x for all applications. Most of the memory overhead results from the shadow memory of ThreadSanitizer used for race detection. The memory overhead of the MUST component in RMA Sanitizer depends on the number of RMA operations performed. It is typically comparably small since the RMA consistency information is only stored when an RMA operation is active and deleted on completion. The higher the number of concurrent RMA operations, the higher is the memory consumption of the consistency tracking. In addition, every concurrent region of an RMA operation is annotated to ThreadSanitizer, which consumes additional memory. Further, the higher the number of processes, the larger are the vector clocks that are sent around between the processes, also leading to a slightly higher memory consumption.

None of the measured applications ran out of memory with RMA Sanitizer for any input size. The detailed results of the memory overhead measurements are included in the thesis artifact described in Appendix A. In general, applications that do not have an excessive memory footprint should be analyzable with RMA Sanitizer.

### Discussion

In summary, RMA Sanitizer shows good scaling behavior for the tested applications, even on large-scale measurements with up to 768 processes. RMA Sanitizer did not report a false race on any of the application runs. The maximum reported slowdown is 27x for NPB BT, while for most applications, it is typically between 1.1x and 10x. The weak scaling benchmarks have a better scaling behavior than strong scaling benchmarks. Three characteristics of an application were identified to influence the overhead: First, the number of local memory accesses instrumented and analyzed by ThreadSanitizer adds a constant overhead to the application, independent of the number of processes. With the static analysis of RMA Sanitizer, this overhead could be significantly reduced for some benchmarks. Second, the higher the number of RMA communication calls the application performs, the more RMA operations must be managed by RMA Sanitizer and annotated to ThreadSanitizer, leading to a higher slowdown. Third, a high number of synchronization operations leads to high pressure on the vector clock exchange, increasing its overhead, as observed for CFD-Proxy. Overall, the results are promising and confirm that RMA Sanitizer is ready to be applied to other applications.

### 6.2.4 Comparison to PARCOACH

In an additional experiment, the overhead of RMA Sanitizer is compared to the overhead of the dynamic on-the-fly race detector implemented in PARCOACH [2, 106, 107]. For PARCOACH, the tool version<sup>19</sup> recommended by its authors is used. PARCOACH is

---

<sup>19</sup><https://gitlab.inria.fr/parcoach/parcoach/-/tree/Merge-non-adjacent+Flush>, commit hash 41584a46259e8178b3c811f97882c19b9103e08d

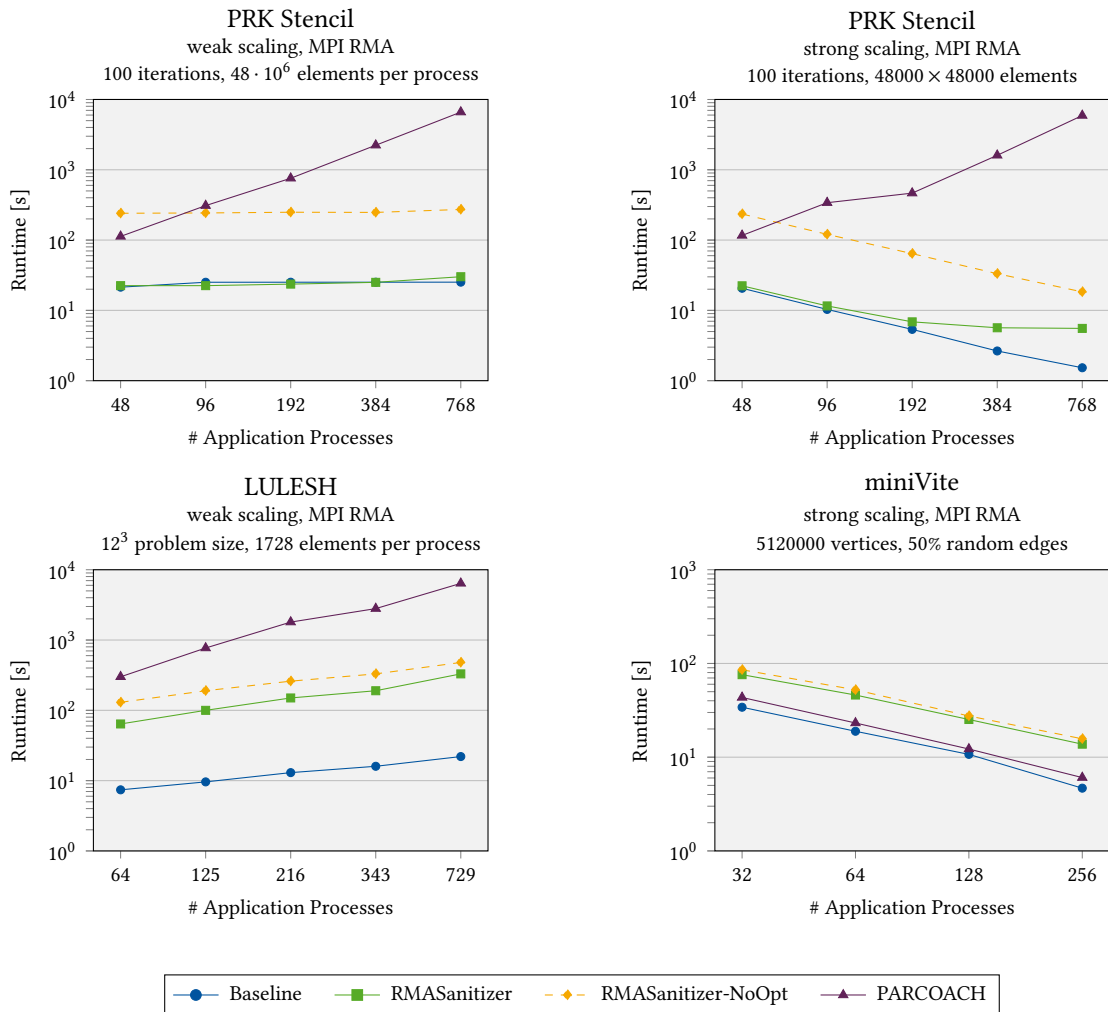


Figure 6.12: Overhead results of RMA Sanitizer and PARCOACH on PRK Stencil, LULESH (ported), and miniVite. The input data sets compared to Figure 6.8 were shrunk to avoid excessive use of cluster resources. The y-axis is scaled logarithmically.

executed with the same setup as RMA Sanitizer, except for the compiler that is required to be Clang 15 for PARCOACH instead of Clang 16 that was used for RMA Sanitizer. The different compiler versions do not affect the baseline performance, as verified by the measurements. The chosen benchmarks for the comparison are PRK Stencil, LULESH, and miniVite. In addition to a weak scaling variant of PRK Stencil, a strong scaling variant of PRK Stencil is included in the measurements. PARCOACH does not support the other applications: NPB BT, as a Fortran application, is incompatible with the tested PARCOACH version. CFD-Proxy is a GASPI application and is also not supported. The execution with miniMD did not terminate. To avoid excessive use of cluster resources, the measurements were only performed once per measurement point. The input data sets for PRK Stencil and LULESH were shrunk compared to Figure 6.8.

Figure 6.12 shows the overhead results of PARCOACH compared to RMA Sanitizer. In PRK Stencil, in both scaling scenarios, PARCOACH has a significantly higher overhead that increases exponentially with the number of processes, starting from a 5x slowdown with 48 processes up to 250x with 768 processes for the weak scaling. A reason for that might be the high amount of local memory accesses in PRK Stencil. PARCOACH does not filter them, so it must analyze them all, as *RMA Sanitizer-NoOpt* does. Another difference to RMA Sanitizer is the way of how memory accesses conflicts are detected: PARCOACH uses a binary search tree to manage the accessed memory intervals. On every memory access, be it a local memory access or an RMA operation, the binary search tree has to be scanned for conflicts and the new memory access has to be inserted. Scanning and inserting a binary search tree requires logarithmic time based on the size of the tree. Instead, the shadow memory approach of RMA Sanitizer has constant access time. As PRK Stencil performs many local memory accesses, this difference in complexity may explain the difference in performance. The experiments in the bachelor thesis of Sem Klauke [47] that replaced the shadow memory approach of ThreadSanitizer with an Interval Skip List, as outlined in Section 5.3.6, also showed orders of magnitudes higher overheads than the shadow memory approach. Another surprising observation is PARCOACH's behavior for the strong scaling variant of PRK Stencil which is nearly the same as that of the weak scaling variant. This may additionally hint at a problem in the tool thread infrastructure that PARCOACH uses to transmit information about ongoing RMA operations.

PARCOACH's behavior on LULESH is similar to that of the PRK stencil, as LULESH also performs a relatively high number of local memory accesses. This benchmark code also tends to be memory-bound, which seems to lead to high overheads in PARCOACH. For miniVite, the overhead results with PARCOACH are different: Here, the number of local memory accesses is lower and the code is rather communication-bound as it has to exchange many neighboring information via *MPI\_Put* and also via *MPI\_Irecv* and *MPI\_Isend*. PARCOACH seems to profit from this pattern, as the overhead is very small with a slowdown of 1.3x for 768 processes, even smaller than with RMA Sanitizer with a slowdown of 2.9x. As PARCOACH does not track the synchronization of individual processes at all, it can completely ignore all *MPI\_Irecv* and *MPI\_Isend* calls. RMA Sanitizer, on the other hand, interprets each call as a synchronization point for the vector clock exchange, leading to a higher overhead.

The PARCOACH authors also used miniVite to compare the results of their approach [107] with MUST-RMA (the predecessor of RMA Sanitizer) in a similar setup. The reported results for MUST-RMA are similar to the results for RMA Sanitizer presented here. However, the PARCOACH authors measured a slightly worse scaling behavior for MUST-RMA. One reason could be that they measured only the time spent in MPI RMA epochs rather than the whole application time. Further, miniVite can be configured to store the generated graphs using either 32-bit or 64-bit elements. In the experiments for this thesis, the 64-bit elements variant led to unreliable results, such as crashes within ThreadSanitizer or false race reports with RMA Sanitizer, which may slow down the execution. Therefore, the 32-bit variant was chosen to produce reliable results, which is also an explanation for the difference to the results of the PARCOACH authors for MUST-RMA.

## 6.3 Results and Discussion

This chapter discusses the classification quality and performance overhead of RMA Sanitizer. The classification quality of RMA Sanitizer was analyzed using the benchmark suite RMA Race Bench, which I developed in the context of my research. RMA Race Bench covers all RMA routines specified in MPI RMA, SHMEM, and GASPI. When designing the test cases, the focus was on covering all possible race scenarios according to the programming model standards, not the support of a particular tool. The test cases were run with RMA Sanitizer, MUST-RMA, and PARCOACH.

On the test cases of RMA Race Bench, RMA Sanitizer has a high accuracy of over 90 percent, showing that it supports all major completion and synchronization mechanisms of MPI RMA, SHMEM, and GASPI. Since accuracy is the most important metric of a correctness checking tool, the results prove that the generalized race detection model works in practice. Furthermore, RMA Sanitizer has a significantly higher accuracy on the RMA Race Bench test cases than the static and dynamic race detectors for MPI RMA implemented in PARCOACH. The main advantage of RMA Sanitizer over the dynamic race detector PARCOACH is the clock-based synchronization tracking that allows in combination with the concurrent region detection a fine-grained reasoning on the concurrency of RMA operations. Even in hybrid race scenarios with OpenMP, RMA Sanitizer correctly classifies nearly all test cases.

The overhead evaluation on different proxy applications with up to 768 processes shows that RMA Sanitizer is applicable to large-scale applications. Depending on the application, the slowdown factor of RMA Sanitizer ranges from 1.1x to 27x. The main factors influencing the slowdown are the number of instrumented local memory accesses and the communication frequency in terms of RMA operations compared to the computation. The higher the number of instrumented and analyzed local memory accesses or the higher the frequency of communication operations, the higher the overhead of RMA Sanitizer. The static analysis used to filter out irrelevant memory accesses in the instrumentation phase proves to be an effective measure for overhead reduction. Its effectiveness depends on the amount of aliasing in the application. Compared to PARCOACH's dynamic race detector, which uses binary search trees to detect conflicting memory accesses, the shadow memory approach utilized by RMA Sanitizer has a smaller overhead due to its constant access time for data race detection.

In summary, the analysis of proxy applications shows that RMA Sanitizer is also applicable to larger real-world applications. Future case studies could include applying RMA Sanitizer on more complex applications such as the computational chemistry toolbox NWChem [104] that partly utilizes MPI RMA [96] for communication.



## 7 Summary and Conclusion

RMA programming models provide an efficient way of communication in modern supercomputers. However, their complexities in terms of consistency and synchronization may lead to data races with undefined behavior. The non-deterministic nature of data races makes it difficult to detect them manually. This thesis addresses the classification, modeling, and on-the-fly detection of data races in RMA programs, focusing on the library-based programming models MPI RMA, OpenSHMEM, and GASPI.

Due to the different ways of ensuring synchronization and consistency in RMA models, various error sources may lead to data races. The classification of RMA races presented in Chapter 2 distinguishes local buffer races, resulting from conflicting memory accesses at the origin process, and remote races, resulting from conflicting memory accesses across processes. The analysis of MPI RMA, OpenSHMEM, and GASPI shows that it is possible to develop a unified description of the synchronization and consistency semantics. This motivates the design of a programming-model-agnostic RMA race detection, as done in this thesis.

To detect data races in RMA programs, process synchronization must be analyzed. Chapter 3 provides a classification of synchronization in distributed-memory programs at the example of MPI, OpenSHMEM, and GASPI. Based on this classification, a set of generalized synchronization primitives has been defined that suffices to capture all relevant process synchronization behavior in RMA models. The concrete synchronization routines of MPI, OpenSHMEM, and GASPI are mapped to the generalized synchronization primitives. The primitives are used to formally define a synchronization tracking mechanism based on an on-the-fly vector clock exchange. The vector clock exchange enables a happened-before analysis of events, which is a prerequisite for runtime data race analysis in RMA programs. It has been implemented in the correctness checking tool MUST and supports MPI, OpenSHMEM, and GASPI. An overhead evaluation with the SPEC MPI 2007 benchmarks shows that the vector clock exchange is applicable to real-world applications with typical slowdowns of 1.1x to 3.5x for up to 768 processes. Besides RMA data race detection, the implemented vector clock exchange has also been used as a building block for MPI file I/O race detection and thread-level concurrency checks for MPI+OpenMP programs.

Chapter 4 provides a formal model to reason on data races in RMA programs. The model extends the happened-before order to a consistency order to incorporate the completion semantics of the RMA models. For that, it defines the semantics of seven generic RMA completion primitives. The primitives serve as an abstraction layer that can represent

the completion behavior of MPI RMA, OpenSHMEM, and GASPI. The application to different RMA codes shows that the model can correctly represent the consistency of the three RMA models. The consistency model is finally combined with the clock-based synchronization model in an RMA race detection model. It records the concurrent region, defining the earliest and latest point in time when an RMA operation's memory access may occur. The presented data race detection algorithm detects overlapping concurrent regions to check for data races. It is the first RMA race detection algorithm using vector clocks to find data races in RMA at runtime.

The RMA race detection model was implemented in an on-the-fly data race detector named RMA Sanitizer, as described in Chapter 5. It combines the existing shared-memory race detector ThreadSanitizer with an RMA operation consistency tracking implemented in the correctness checking tool MUST. RMA Sanitizer is the first on-the-fly detector for RMA programs supporting different RMA programming models. It understands the most extensive set of synchronization and consistency mechanisms compared to related work. Currently, RMA Sanitizer supports three RMA models: MPI RMA, SHMEM, and GASPI. Due to its modular architecture, supporting further RMA models only requires adding a wrapper module that maps the concrete routine to the abstract synchronization and completion primitives of the race detection model.

Lastly, the classification quality and the overhead evaluation presented in Chapter 6 prove that RMA Sanitizer achieves a high detection accuracy with low overhead, even for large-scale application runs with up to 768 processes. For the classification quality analysis, the benchmark suite RMA RaceBench has been developed in the context of this thesis. It is a semantically-driven test suite that provides tests with different RMA race scenarios covering all RMA communication and synchronization routines defined in MPI RMA, OpenSHMEM, and GASPI. The RMA RaceBench tests systematically iterate through different categories challenging the accuracy of RMA race detectors to detect conflicts, missing synchronization and consistency, incorrect atomicity, and also covers hybrid data race scenarios. RMA RaceBench also comes with a test harness that automatically classifies the test results of the tools. On this benchmark suite, RMA Sanitizer shows a high detection accuracy, proving that it supports all major completion and synchronization mechanisms of RMA models and outperforms existing related RMA race detectors. It achieved an accuracy of over 90 percent, which is significantly higher than the accuracy of the static and dynamic race detectors in PAROCACH, which achieved 60 percent. The overhead evaluation on different proxy applications showed that the introduced slowdown of RMA Sanitizer ranges from 1.1x to 27x, depending on the application. A static analysis integrated into RMA Sanitizer that filters out irrelevant memory accesses helps to keep its analysis overhead low.

In summary, this thesis provided key insights into the challenge of portable data race detection in RMA applications. It defined a generalized formal model and novel methods for race detection in RMA models, resulting in an accurate and scalable race detection tool.

## 7.1 Future Work

Future work could extend the general race detection model to further RMA programming models. Supporting new programming models only requires mapping the concrete RMA routines to the abstract model primitives. The current implementation of RMA Sanitizer assumes library-based RMA models whose routines can be intercepted at runtime. An extension to support RMA languages or directive-based approaches only requires adaptations in the instrumentation mechanism.

The presented race detection model assumes an execution with single-threaded processes. Extending the model to fully support hybrid models combining RMA with multithreading, such as OpenMP, significantly complicates the reasoning on RMA data races. Although RMA Sanitizer can correctly detect some data races with hybrid parallelism, the current race detection model overapproximates synchronization and might miss data races. This thesis presented some initial ideas on capturing the hybrid parallelism in the vector clocks, which would have to be integrated into the model.

RMA Sanitizer has been proven to work on proxy applications running with a higher number of processes. An overhead breakdown of RMA Sanitizer's main components has shown that the analysis bottleneck varies depending on the application. In future evaluations, external performance analysis tools might be used to examine the overhead of the individual analysis components more thoroughly. However, using a performance analysis tool to investigate the overhead of a correctness tool is challenging as the instrumentation of both tools may interfere with each other.

The presented race detection model has been designed to detect races in RMA models intended to run on CPUs. In recent years, RMA models for intra-kernel communication of data across GPUs have been developed. As GPUs are more constrained regarding instrumentation and available memory, another future research direction would be to investigate how well RMA Sanitizer's race detection approach matches GPU architectures.



## 8 Statement of Originality

The research of this thesis was conducted at the Chair for High Performance Computing (HPC), headed by Professor Matthias S. Müller, in the computer science department of RWTH Aachen University. It profited substantially from the supervision of Professor Müller, the advice of the HPC group leader, Christian Terboven, and the discussion with team members at the HPC chair. Moreover, the guidance from my colleague Joachim Jenke (né Protze), who served as an additional research mentor throughout the whole time, helped a lot in further shaping my research.

The thesis contents are partly based on previous publications with several authors. In the following, I will list and detail my contributions to those previously published research works. The author order of the publications represents their contributions in terms of novel ideas.

- **An On-the-Fly Method to Exchange Vector Clocks in Distributed-Memory Programs**, Schwitanski et al. [89]: This work introduces the generic clock-based synchronization model for MPI RMA, OpenSHMEM, and GASPI, including a prototype implementation for MPI RMA and OpenSHMEM. The original idea and methodology of the vector clock exchange in RMA programs were proposed by me. Felix Tomski designed an initial version of the generalized synchronization model for distributed-memory programs and implemented it in MUST as part of his master thesis [103] that I supervised. For the subsequent publication in the research paper, I further refined this model and extended the implementation prototyped by Felix Tomski to also work with OpenSHMEM in addition to MPI RMA, and I re-evaluated some of the results on the SPEC MPI 2007 benchmarks. Chapter 3 (in particular Sections 3.2, 3.3, 3.4, 3.5) of this thesis is partly based on this publication.
- **On-the-Fly Data Race Detection for MPI RMA Programs with MUST**, Schwitanski et al. [90]: In this work, I present the RMA race detection approach specifically for MPI RMA and the corresponding race detection tool MUST-RMA. I designed the race detection model, implemented it in MUST, and evaluated it on simple MPI RMA kernels. This work is partly based on the RMA race detection approach which I have developed and implemented in my master thesis [88], supervised by Joachim Jenke. Felix Tomski and Joachim Jenke provided feedback on the ideas and methodology presented in the paper. The approach of this paper has been generalized to work with OpenSHMEM and GASPI in a follow-up publication [93] presenting RMA Sanitizer. The concurrent region detection approach presented in

Section 4.2 and the resulting implementation discussed in Section 5.3 of this thesis is partly based on that publication.

- **RMARaceBench: A Microbenchmark Suite to Evaluate Race Detection Tools for RMA Programs**, Schwitanski et al. [91]: This publication introduces the classification quality benchmark suite RMARaceBench comprising data race test cases written in MPI RMA, OpenSHMEM, and GASPI. It presents and compares the classification quality results of different RMA race detectors. The main conceptual design, implementation of most test cases, and evaluation were done by me. Sven Klotz helped with designing the OpenSHMEM and the hybrid OpenMP test cases. RMARaceBench is used in Section 6.1 to evaluate the classification quality of RMA Sanitizer.
- **Mapping High-Level Concurrency from OpenMP and MPI to ThreadSanitizer Fibers**, Jenke et al. [45]: This work showcases different use cases of ThreadSanitizer fiber annotations such as data race detection in OpenMP tasking, MPI non-blocking communication, and MPI RMA. Joachim Jenke contributed as the primary author the main use cases and a performance evaluation of the fiber annotations. I contributed an explanation of the MPI RMA race detection use case for fiber annotations. The fiber annotations are one of the main building blocks of RMA Sanitizer. Section 5.2.4 and Section 5.3.4 are partly based on this publication.
- **Leveraging Static Analysis to Accelerate Dynamic Race Detection for Remote Memory Access Programs**, Schwitanski et al. [92]: This work presents static analysis techniques to reduce the overhead of RMA race detection by instrumenting only relevant local memory accesses. I proposed the initial idea of exploring static analysis techniques for that purpose. Yussur Mustafa Oraji designed, implemented, and initially evaluated different static analysis techniques in his bachelor thesis [72], supervised by me. He continued improving the implementation as a student worker under my supervision. This includes adding a category to the RMARaceBench suite to specifically evaluate the effect of the static analysis on the RMA race detection accuracy. For the subsequent publication of the results in this research paper, I added a detailed evaluation of the overhead reduction due to the different static analysis techniques. Section 5.3.1 is based on this work.
- **RMA Sanitizer: Generalized Runtime Detection of Data Races in Remote Memory Access Applications**, Schwitanski et al. [93]: In this work, the generalized runtime race detector RMA Sanitizer for MPI RMA, OpenSHMEM, and GASPI is presented. This work defines a generic race detection model for RMA, which is an extension of my previous work in [90] and describes the corresponding implementation named RMA Sanitizer. I formalized the generalized race detection model, extended MUST-RMA to RMA Sanitizer, and evaluated its classification quality and overhead. Some applications used for the overhead evaluation were manually ported to MPI RMA by Cornelius Pätzold, a student worker supervised by me. Further, the overhead benchmark suite developed specifically for RMA Sanitizer was a collaborative effort of Cornelius Pätzold, Yussur Mustafa Oraji, and me. The

inference rule system and concurrent region approach presented in Chapter 4, the description of RMA Sanitizer's tool architecture in Section 5.3, and the classification quality and overhead evaluation in Chapter 6 is partly based on this work.

- **Correctness Checking of MPI+OpenMP Applications Using Vector Clocks in MUST**, Pätzold et al. [76]: This work presents the application of an OpenMP vector clock exchange to check for errors in MPI+OpenMP applications with MUST. Cornelius Pätzold implemented these checks in his master thesis [75] that I supervised. Cornelius Pätzold also contributed the main paragraphs to the paper under my supervision. The explanations on the thread-level concurrency checks in Section 3.7.2 are partly based on this work.
- **MPI-BugBench: A Framework for Assessing MPI Correctness Tools**, Jammer et al. [44]: This work is a collaborative effort between TU Darmstadt (TUDa), INRIA in Bordeaux, and RWTH Aachen University. The proposed classification benchmark suite, MPI-BugBench, is a combination of the independently developed suites MPI-Corrbench [59], MPI Bugs Initiative [57], and RMA RaceBench [91]. Tim Jammer (TUDa) improved the test generation infrastructure to increase the coverage of the benchmark suite and integrated all MPI point-to-point tests. Emmanuelle Saillard (INRIA) worked on the MPI collective tests. I added tests for MPI RMA to MPI-BugBench based on my experience of designing RMA RaceBench. Radjasouria Vinayagame (INRIA) verified and improved the RMA test cases. The preparation and execution of the evaluation were done by Emmanuelle Saillard (INRIA) and me.



# A Thesis Artifact

The thesis artifact is available at the following URL:

<https://doi.org/10.18154/RWTH-2024-11181>

The artifact contains the source code of the RMA Sanitizer implementation described in Chapter 5 under a BSD 3-clause license. For the reproducibility of the classification quality evaluation in Section 6.1, the RMA Race Bench suite and all results of the different tools are included. All overhead measurements utilize the JUBE Benchmarking Environment for reproducible measurements. The measurement infrastructure and the results of the SPEC MPI 2007 overhead measurements in Section 3.5 are included. Also, the complete measurement infrastructure for reproducing the RMA Sanitizer and PARCOACH overhead measurements and all results used in Section 6.2.3 and Section 6.2.4 are part of the thesis artifact. Further information and instructions on reproducibility are provided in the artifact description.



# B RMARaceBench Results

The following tables list the test cases and detailed results of the RMA race detectors on the MPI RMA, SHMEM, and GASPI test cases of RMARaceBench. The setup has been explained in detail in Section 6.1.3.

The categorization results are abbreviated as follows: TP = True Positive, TN = True Negative, FP = False Positive, FN = False Negative, TO = Timeout.

## B.1 MPI RMA Results

|  | PARCOACH-<br>static | PARCOACH-<br>dynamic | MUST-RMA | RMASanitizer | RMASanitizer-<br>NoOpt |
|--|---------------------|----------------------|----------|--------------|------------------------|
| 001-MPI-conflict-put-load-local-no.c           | TN                  | TN                   | TN       | TN           | TN                     |
| 002-MPI-conflict-put-store-local-yes.c         | TP                  | TP                   | TP       | TP           | TP                     |
| 003-MPI-conflict-put-put-local-no.c            | TN                  | TN                   | TN       | TN           | TN                     |
| 004-MPI-conflict-get-load-local-yes.c          | TP                  | TP                   | TP       | TP           | TP                     |
| 005-MPI-conflict-get-store-local-yes.c         | TP                  | TP                   | TP       | TP           | TP                     |
| 006-MPI-conflict-get-put-local-yes.c           | TP                  | TP                   | TP       | TP           | TP                     |
| 007-MPI-conflict-get-get-local-yes.c           | TP                  | TP                   | TP       | TP           | TP                     |
| 008-MPI-conflict-acc-store-local-yes.c         | FN                  | TP                   | TP       | TP           | TP                     |
| 009-MPI-conflict-acc-load-local-no.c           | TN                  | TN                   | TN       | TN           | TN                     |
| 010-MPI-conflict-gacc-store-local-yes.c        | FN                  | FN                   | TP       | TP           | TP                     |
| 011-MPI-conflict-gacc-load-local-yes.c         | FN                  | FN                   | TP       | TP           | TP                     |
| 012-MPI-conflict-fop-store-local-yes.c         | FN                  | FN                   | FN       | TP           | TP                     |
| 013-MPI-conflict-fop-load-local-yes.c          | FN                  | FN                   | FN       | TP           | TP                     |
| 014-MPI-conflict-cas-store-local-yes.c         | FN                  | FN                   | FN       | TP           | TP                     |
| 015-MPI-conflict-cas-load-local-yes.c          | FN                  | FN                   | FN       | TP           | TP                     |
| 016-MPI-conflict-get-load-remote-no.c          | -                   | TN                   | TN       | TN           | TN                     |
| 017-MPI-conflict-get-get-remote-no.c           | -                   | TN                   | TN       | TN           | TN                     |
| 018-MPI-conflict-get-store-remote-yes.c        | -                   | FN                   | TP       | TP           | TP                     |
| 019-MPI-conflict-get-put-remote-yes.c          | -                   | TP                   | TP       | TP           | TP                     |
| 020-MPI-conflict-get-gaccread-remote-no.c      | -                   | TN                   | TN       | TN           | TN                     |
| 021-MPI-conflict-get-acc-remote-yes.c          | -                   | TP                   | FN       | TP           | TP                     |
| 022-MPI-conflict-put-load-remote-yes.c         | -                   | TP                   | TP       | TP           | TP                     |
| 023-MPI-conflict-put-store-remote-yes.c        | -                   | TP                   | TP       | TP           | TP                     |
| 024-MPI-conflict-put-put-remote-yes.c          | -                   | TP                   | TP       | TP           | TP                     |
| 025-MPI-conflict-put-gaccread-remote-yes.c     | -                   | FN                   | FN       | TP           | TP                     |
| 026-MPI-conflict-put-acc-remote-yes.c          | -                   | TP                   | FN       | TP           | TP                     |
| 027-MPI-conflict-acc-load-remote-yes.c         | -                   | TP                   | FN       | TP           | TP                     |
| 028-MPI-conflict-acc-store-remote-yes.c        | -                   | TP                   | FN       | TP           | TP                     |
| 029-MPI-conflict-acc-acc-remote-no.c           | -                   | FP                   | TN       | TN           | TN                     |
| 030-MPI-conflict-acc-gaccread-remote-no.c      | -                   | TN                   | TN       | TN           | TN                     |
| 031-MPI-conflict-gaccread-gaccread-remote-no.c | -                   | TN                   | TN       | TN           | TN                     |
| 032-MPI-conflict-gaccread-load-remote-no.c     | -                   | TN                   | TN       | TN           | TN                     |

## B RMARaceBench Results

|  | PARCOACH-<br>static | PARCOACH-<br>dynamic | MUST-RMA | RMASanitizer | RMASanitizer-<br>NoOpt |
|--|---------------------|----------------------|----------|--------------|------------------------|
| 033-MPI-conflict-gaccread-store-remote-yes.c         | -                   | FN                   | FN       | TP           | TP                     |
| 034-MPI-conflict-gacc-store-remote-yes.c             | -                   | FN                   | FN       | TP           | TP                     |
| 035-MPI-conflict-gacc-gacc-remote-no.c               | -                   | TN                   | TN       | TN           | TN                     |
| 036-MPI-conflict-fop-fop-remote-no.c                 | -                   | TN                   | TN       | TN           | TN                     |
| 037-MPI-conflict-fop-store-remote-yes.c              | -                   | FN                   | FN       | TP           | TP                     |
| 038-MPI-conflict-cas-store-remote-yes.c              | -                   | FN                   | FN       | TP           | TP                     |
| 039-MPI-conflict-cas-cas-remote-no.c                 | -                   | TN                   | TN       | TN           | TN                     |
| 001-MPI-sync-fence-local-yes.c                       | TP                  | TP                   | TP       | TP           | TP                     |
| 002-MPI-sync-fence-local-no.c                        | TN                  | TN                   | TN       | TN           | TN                     |
| 003-MPI-sync-lock-local-yes.c                        | TP                  | TO                   | TP       | TP           | TP                     |
| 004-MPI-sync-lock-local-no.c                         | FP                  | TO                   | TN       | TN           | TN                     |
| 005-MPI-sync-lock-flush-local-yes.c                  | TP                  | TO                   | TP       | TP           | TP                     |
| 006-MPI-sync-lock-flush-local-no.c                   | TN                  | TO                   | TN       | TN           | TN                     |
| 007-MPI-sync-lockall-flushlocalall-local-yes.c       | TP                  | TP                   | TP       | TP           | TP                     |
| 008-MPI-sync-lockall-flushlocalall-local-no.c        | FP                  | FP                   | TN       | TN           | TN                     |
| 009-MPI-sync-request-local-yes.c                     | FN                  | FN                   | TP       | TP           | TP                     |
| 010-MPI-sync-request-local-no.c                      | TN                  | TN                   | TN       | TN           | TN                     |
| 011-MPI-sync-pscw-local-yes.c                        | TP                  | FN                   | TP       | TP           | TP                     |
| 012-MPI-sync-pscw-local-no.c                         | FP                  | TN                   | TN       | TN           | TN                     |
| 013-MPI-sync-lockall-flushall-remote-no.c            | -                   | TN                   | TN       | TN           | TN                     |
| 014-MPI-sync-lockall-flushall-remote-yes.c           | -                   | FN                   | TP       | TP           | TP                     |
| 015-MPI-sync-lockall-barrier-remote-no.c             | -                   | TN                   | TN       | TN           | TN                     |
| 016-MPI-sync-lockall-barrier-remote-yes.c            | -                   | FN                   | TP       | TP           | TP                     |
| 017-MPI-sync-lockall-remote-yes.c                    | -                   | TP                   | TP       | TP           | TP                     |
| 018-MPI-sync-fence-3procs-remote-yes.c               | -                   | TP                   | TP       | TP           | TP                     |
| 019-MPI-sync-fence-3procs-remote-no.c                | -                   | TN                   | TN       | TN           | TN                     |
| 020-MPI-sync-lock-barrier-nonconsistent-remote-yes.c | -                   | TO                   | TP       | TP           | TP                     |
| 021-MPI-sync-lock-barrier-remote-yes.c               | -                   | TO                   | TP       | TP           | TP                     |
| 022-MPI-sync-lock-barrier-remote-no.c                | -                   | TO                   | TN       | TN           | TN                     |
| 023-MPI-sync-lock-barrier-sameorigin-remote-no.c     | -                   | TO                   | TN       | TN           | TN                     |
| 024-MPI-sync-lock-barrier-sameorigin-remote-yes.c    | -                   | TO                   | TP       | TP           | TP                     |
| 025-MPI-sync-lock-flushlocal-sameorigin-remote-yes.c | -                   | TP                   | TP       | TP           | TP                     |
| 026-MPI-sync-lock-flushlocal-sameorigin-remote-no.c  | -                   | FP                   | TN       | TN           | TN                     |
| 027-MPI-sync-lock-exclusive-remote-no.c              | -                   | TO                   | TN       | TN           | TN                     |
| 028-MPI-sync-lock-exclusive-3procs-remote-no.c       | -                   | TO                   | TN       | TN           | TN                     |
| 029-MPI-sync-lock-exclusive-remote-yes.c             | -                   | TO                   | TP       | TP           | TP                     |
| 030-MPI-sync-lock-sendrecv-remote-yes.c              | -                   | TO                   | TP       | TP           | TP                     |
| 031-MPI-sync-lock-sendrecv-remote-no.c               | -                   | TO                   | TN       | TN           | TN                     |
| 032-MPI-sync-lock-sendrecv-3procs-remote-no.c        | -                   | TO                   | FP       | TN           | TN                     |
| 033-MPI-sync-lock-sendrecv-3procs-remote-yes.c       | -                   | TO                   | TP       | TP           | TP                     |
| 034-MPI-sync-pscw-remote-no.c                        | -                   | TN                   | TN       | TN           | TN                     |
| 035-MPI-sync-pscw-remote-yes.c                       | -                   | FN                   | TP       | TP           | TP                     |
| 036-MPI-sync-polling-remote-yes.c                    | -                   | TO                   | TP       | TP           | TP                     |
| 037-MPI-sync-lock-ordering-remote-yes.c              | -                   | TO                   | FN       | FN           | FN                     |
| 001-MPI-atomic-customdatatype-remote-no.c            | -                   | FP                   | TN       | TN           | TN                     |
| 002-MPI-atomic-customdatatype-remote-yes.c           | -                   | TP                   | FN       | TP           | TP                     |
| 003-MPI-atomic-disp-remote-yes.c                     | -                   | FN                   | FN       | TP           | TP                     |
| 004-MPI-atomic-disp-remote-no.c                      | -                   | FP                   | TN       | TN           | TN                     |
| 005-MPI-atomic-short-int-remote-yes.c                | -                   | TP                   | FN       | TP           | TP                     |
| 006-MPI-atomic-float-int-remote-yes.c                | -                   | FN                   | FN       | TP           | TP                     |
| 007-MPI-atomic-float-int-sameorigin-remote-yes.c     | -                   | FN                   | FN       | TP           | TP                     |
| 008-MPI-atomic-double-float-remote-yes.c             | -                   | FN                   | FN       | TP           | TP                     |
| 009-MPI-atomic-int-int-remote-no.c                   | -                   | FP                   | TN       | TN           | TN                     |
| 010-MPI-atomic-int-int-sameorigin-remote-no.c        | -                   | FP                   | TN       | TN           | TN                     |
| 001-MPI-hybrid-master-local-yes.c                    | TP                  | TP                   | TP       | TP           | TP                     |
| 002-MPI-hybrid-master-local-no.c                     | FP                  | TO                   | TN       | TN           | TN                     |

|   | PARCOACH-<br>static | PARCOACH-<br>dynamic | MUST-RMA | RMASanitizer | RMASanitizer-<br>NoOpt |
|---|---------------------|----------------------|----------|--------------|------------------------|
| 003-MPI-hybrid-single-local-yes.c                   | TP                  | TP                   | TP       | TP           | TP                     |
| 004-MPI-hybrid-single-local-no.c                    | FP                  | TO                   | TN       | TN           | TN                     |
| 005-MPI-hybrid-ordered-local-no.c                   | FP                  | TO                   | TN       | TN           | TN                     |
| 006-MPI-hybrid-for-local-yes.c                      | TP                  | TP                   | TP       | TP           | TP                     |
| 007-MPI-hybrid-section-local-yes.c                  | TP                  | TP                   | TP       | TP           | TP                     |
| 008-MPI-hybrid-section-local-no.c                   | FP                  | TO                   | TN       | TN           | TN                     |
| 009-MPI-hybrid-task-local-yes.c                     | FN                  | TO                   | TP       | TP           | TP                     |
| 010-MPI-hybrid-task-local-no.c                      | FP                  | TO                   | TN       | TN           | TN                     |
| 011-MPI-hybrid-master-remote-yes.c                  | -                   | TO                   | TP       | TP           | TP                     |
| 012-MPI-hybrid-master-remote-no.c                   | -                   | TO                   | TN       | TN           | TN                     |
| 013-MPI-hybrid-single-remote-yes.c                  | -                   | TO                   | TP       | TP           | TP                     |
| 014-MPI-hybrid-single-remote-no.c                   | -                   | TO                   | TN       | TN           | TN                     |
| 015-MPI-hybrid-task-remote-yes.c                    | -                   | TO                   | TP       | TP           | TP                     |
| 016-MPI-hybrid-task-remote-no.c                     | -                   | TO                   | TN       | TN           | TN                     |
| 017-MPI-hybrid-section-remote-yes.c                 | -                   | TO                   | TP       | TP           | TP                     |
| 018-MPI-hybrid-section-remote-no.c                  | -                   | TO                   | TN       | TN           | TN                     |
| 019-MPI-hybrid-ordered-remote-no.c                  | -                   | TO                   | TN       | TN           | TN                     |
| 020-MPI-hybrid-for-remote-yes.c                     | -                   | TO                   | TP       | TP           | TP                     |
| 021-MPI-hybrid-section-barrier-origin-remote-yes.c  | -                   | TO                   | FN       | FN           | FN                     |
| 022-MPI-hybrid-section-sendrecv-origin-remote-yes.c | -                   | TO                   | FN       | FN           | FN                     |
| 001-MPI-misc-put-load-deep-nesting-local-no.c       | TN                  | TN                   | TN       | TN           | TN                     |
| 002-MPI-misc-put-load-aliasing-local-no.c           | TN                  | TN                   | TN       | TN           | TN                     |
| 003-MPI-misc-put-load-retval-local-no.c             | TN                  | TN                   | TN       | TN           | TN                     |
| 004-MPI-misc-put-load-memcpy-local-no.c             | TN                  | TN                   | TN       | TN           | TN                     |
| 005-MPI-misc-get-load-deep-nesting-local-yes.c      | FN                  | FN                   | TP       | TP           | TP                     |
| 006-MPI-misc-get-load-aliasing-local-yes.c          | TP                  | TP                   | TP       | TP           | TP                     |
| 007-MPI-misc-get-load-retval-local-yes.c            | TP                  | TP                   | TP       | TP           | TP                     |
| 008-MPI-misc-get-load-memcpy-local-yes.c            | TP                  | TP                   | TP       | FN           | TP                     |
| 009-MPI-misc-get-load-deep-nesting-remote-no.c      | -                   | TN                   | TN       | TN           | TN                     |
| 010-MPI-misc-get-load-funcpointer-remote-no.c       | -                   | TO                   | TN       | TN           | TN                     |
| 011-MPI-misc-get-load-aliasing-remote-no.c          | -                   | TN                   | TN       | TN           | TN                     |
| 012-MPI-misc-get-load-retval-remote-no.c            | -                   | TN                   | TN       | TN           | TN                     |
| 013-MPI-misc-get-load-memcpy-remote-no.c            | -                   | TO                   | TN       | TN           | TN                     |
| 014-MPI-misc-get-store-deep-nesting-remote-yes.c    | -                   | FN                   | TP       | TP           | TP                     |
| 015-MPI-misc-get-store-funcpointer-remote-yes.c     | -                   | FN                   | TP       | FN           | TP                     |
| 016-MPI-misc-get-store-aliasing-remote-yes.c        | -                   | FN                   | TP       | TP           | TP                     |
| 017-MPI-misc-get-store-retval-remote-yes.c          | -                   | FN                   | TP       | TP           | TP                     |
| 018-MPI-misc-get-store-memcpy-remote-yes.c          | -                   | FN                   | TP       | FN           | TP                     |

## B.2 SHMEM Results

|   | RMASanitizer | RMASanitizer-<br>NoOpt |
|---|--------------|------------------------|
| 001-shmem-conflict-putnbi-load-local-no.c           | TN           | TN                     |
| 002-shmem-conflict-putnbi-store-local-yes.c         | TP           | TP                     |
| 003-shmem-conflict-putnbi-putnbi-local-no.c         | TN           | TN                     |
| 004-shmem-conflict-getnbi-load-local-yes.c          | TP           | TP                     |
| 005-shmem-conflict-getnbi-store-local-yes.c         | TP           | TP                     |
| 006-shmem-conflict-getnbi-putnbi-local-yes.c        | TP           | TP                     |
| 007-shmem-conflict-getnbi-getnbi-local-yes.c        | TP           | TP                     |
| 008-shmem-conflict-put_signal_nbi-store-local-yes.c | TP           | TP                     |
| 009-shmem-conflict-put_signal_nbi-load-local-no.c   | TN           | TN                     |
| 010-shmem-conflict-atomicfetchnbi-store-local-yes.c | TP           | TP                     |
| 011-shmem-conflict-atomicfetchnbi-load-local-yes.c  | TP           | TP                     |

## B RMARaceBench Results

|  | RMASanitizer | RMASanitizer-<br>NoOpt |
|--|--------------|------------------------|
| 012-shmem-conflict-atomicfetchincnbi-store-local-yes.c             | TP           | TP                     |
| 013-shmem-conflict-atomicfetchincnbi-load-local-yes.c              | TP           | TP                     |
| 014-shmem-conflict-atomiccompareswapnbi-load-local-yes.c           | TP           | TP                     |
| 015-shmem-conflict-atomiccompareswapnbi-store-local-yes.c          | TP           | TP                     |
| 016-shmem-conflict-get-load-remote-no.c                            | TN           | TN                     |
| 017-shmem-conflict-get-get-remote-no.c                             | TN           | TN                     |
| 018-shmem-conflict-get-store-remote-yes.c                          | TP           | TP                     |
| 019-shmem-conflict-get-put-remote-yes.c                            | TP           | TP                     |
| 020-shmem-conflict-get-atomicfetch-remote-no.c                     | TN           | TN                     |
| 021-shmem-conflict-get-atomicset-remote-yes.c                      | TP           | TP                     |
| 022-shmem-conflict-put-load-remote-yes.c                           | TP           | TP                     |
| 023-shmem-conflict-put-store-remote-yes.c                          | TP           | TP                     |
| 024-shmem-conflict-put-put-remote-yes.c                            | TP           | TP                     |
| 025-shmem-conflict-put-atomicfetch-remote-yes.c                    | TP           | TP                     |
| 026-shmem-conflict-put-atomicset-remote-yes.c                      | TP           | TP                     |
| 027-shmem-conflict-atomicset-load-remote-yes.c                     | TP           | TP                     |
| 028-shmem-conflict-atomicset-store-remote-yes.c                    | TP           | TP                     |
| 029-shmem-conflict-atomicset-atomicset-remote-no.c                 | TN           | TN                     |
| 030-shmem-conflict-atomicset-atomicfetch-remote-no.c               | TN           | TN                     |
| 031-shmem-conflict-atomicfetch-atomicfetch-remote-no.c             | TN           | TN                     |
| 032-shmem-conflict-atomicfetch-load-remote-no.c                    | TN           | TN                     |
| 033-shmem-conflict-atomicfetch-store-remote-yes.c                  | TP           | TP                     |
| 034-shmem-conflict-put_signal-store-remote-yes.c                   | TP           | TP                     |
| 035-shmem-conflict-put_signal-put_signal-remote-yes.c              | TP           | TP                     |
| 036-shmem-conflict-g-store-remote-yes.c                            | TP           | TP                     |
| 037-shmem-conflict-g-put-remote-yes.c                              | TP           | TP                     |
| 038-shmem-conflict-p-load-remote-yes.c                             | TP           | TP                     |
| 039-shmem-conflict-p-get-remote-yes.c                              | TP           | TP                     |
| 040-shmem-conflict-iput-store-remote-yes.c                         | TP           | TP                     |
| 041-shmem-conflict-iput-put-remote-yes.c                           | TP           | TP                     |
| 042-shmem-conflict-iget-store-remote-yes.c                         | TP           | TP                     |
| 043-shmem-conflict-iget-put-remote-yes.c                           | TP           | TP                     |
| 044-shmem-conflict-atomicfetchnbi-atomicfetchnbi-remote-no.c       | TN           | TN                     |
| 045-shmem-conflict-atomicfetchnbi-load-remote-no.c                 | TN           | TN                     |
| 046-shmem-conflict-atomicfetchinc-atomicfetchinc-remote-no.c       | TN           | TN                     |
| 047-shmem-conflict-atomicfetchnbi-store-remote-yes.c               | TP           | TP                     |
| 048-shmem-conflict-atomiccompareswapnbi-store-remote-yes.c         | TP           | TP                     |
| 049-shmem-conflict-atomiccompareswapnbi-atomicfetchnbi-remote-no.c | TN           | TN                     |
| <hr/>  |              |                        |
| 001-shmem-sync-barrierall-local-yes.c                              | TP           | TP                     |
| 002-shmem-sync-barrierall-local-no.c                               | TN           | TN                     |
| 003-shmem-sync-quiet-local-yes.c                                   | TP           | TP                     |
| 004-shmem-sync-quiet-local-no.c                                    | TN           | TN                     |
| 005-shmem-sync-waituntil-local-no.c                                | TN           | TN                     |
| 006-shmem-sync-waituntil-local-yes.c                               | FN           | FN                     |
| 007-shmem-sync-barrierall-remote-yes.c                             | TP           | TP                     |
| 008-shmem-sync-barrierall-remote-no.c                              | TN           | TN                     |
| 009-shmem-sync-quiet-sync-remote-no.c                              | TN           | TN                     |
| 010-shmem-sync-quiet-sync-remote-yes.c                             | TP           | TP                     |
| 011-shmem-sync-fence-put-put-remote-no.c                           | TN           | TN                     |
| 012-shmem-sync-fence-getnbi-put-remote-yes.c                       | TP           | TP                     |
| 013-shmem-sync-lock-remote-no.c                                    | TN           | TN                     |
| 014-shmem-sync-lock-remote-yes.c                                   | TP           | TP                     |
| 015-shmem-sync-waituntil-remote-yes.c                              | TP           | TP                     |
| 016-shmem-sync-waituntil-remote-no.c                               | TN           | TN                     |
| 017-shmem-sync-putsignal-remote-no.c                               | FP           | FP                     |
| 018-shmem-sync-putsignal-remote-yes.c                              | TP           | TP                     |
| 019-shmem-sync-ctx-remote-no.c                                     | TN           | TN                     |
| 020-shmem-sync-ctx-remote-yes.c                                    | TP           | TP                     |
| 021-shmem-sync-collective-reduce-remote-no.c                       | TN           | TN                     |
| 022-shmem-sync-collective-reduce-remote-yes.c                      | TP           | TP                     |
| 023-shmem-sync-team-sync-remote-yes.c                              | TP           | TP                     |
| 024-shmem-sync-team-sync-remote-no.c                               | TN           | TN                     |
| <hr/>  |              |                        |
| 001-shmem-atomic-different-ctx-remote-yes.c                        | FN           | FN                     |
| 002-shmem-atomic-same-ctx-remote-no.c                              | TN           | TN                     |
| 003-shmem-atomic-same-domain-remote-no.c                           | TN           | TN                     |
| 004-shmem-atomic-int-int-remote-no.c                               | TN           | TN                     |

|  | RMASanitizer | RMASanitizer-<br>NoOpt |
|--|--------------|------------------------|
| 005-shmem-atomic-int-int-sameorigin-remote-no.c                  | TN           | TN                     |
| 006-shmem-atomic-double-long-remote-yes.c                        | TP           | TP                     |
| 007-shmem-atomic-int-long-remote-yes.c                           | TP           | TP                     |
| 008-shmem-atomic-int-float-remote-yes.c                          | TP           | TP                     |
| 009-shmem-atomic-int-float-sameorigin-remote-yes.c               | TP           | TP                     |
| 001-shmem-hybrid-for-local-yes.c                                 | TP           | TP                     |
| 002-shmem-hybrid-for-ordered-local-no.c                          | TN           | TN                     |
| 003-shmem-hybrid-master-local-no.c                               | TN           | TN                     |
| 004-shmem-hybrid-master-local-yes.c                              | TP           | TP                     |
| 005-shmem-hybrid-single-local-no.c                               | TN           | TN                     |
| 006-shmem-hybrid-single-local-yes.c                              | TP           | TP                     |
| 007-shmem-hybrid-section-local-no.c                              | TN           | TN                     |
| 008-shmem-hybrid-section-local-yes.c                             | TP           | TP                     |
| 009-shmem-hybrid-task-local-no.c                                 | TN           | TN                     |
| 010-shmem-hybrid-task-local-yes.c                                | FN           | TP                     |
| 011-shmem-hybrid-for-ordered-remote-no.c                         | TN           | TN                     |
| 012-shmem-hybrid-for-remote-yes.c                                | TP           | TP                     |
| 013-shmem-hybrid-master-remote-no.c                              | TN           | TN                     |
| 014-shmem-hybrid-master-remote-yes.c                             | TP           | TP                     |
| 015-shmem-hybrid-single-remote-no.c                              | TN           | TN                     |
| 016-shmem-hybrid-single-remote-yes.c                             | TP           | TP                     |
| 017-shmem-hybrid-section-remote-no.c                             | TN           | TN                     |
| 018-shmem-hybrid-section-remote-yes.c                            | TP           | TP                     |
| 019-shmem-hybrid-task-remote-no.c                                | TN           | TN                     |
| 020-shmem-hybrid-task-remote-yes.c                               | TP           | TP                     |
| 021-shmem-hybrid-lock-section-barrier-origin-remote-yes.c        | FN           | FN                     |
| 022-shmem-hybrid-lock-section-barrier-origin-signal-remote-yes.c | FN           | FN                     |
| 001-shmem-misc-putnbi-load-deep-nesting-local-no.c               | TN           | TN                     |
| 002-shmem-misc-putnbi-load-aliasing-local-no.c                   | TN           | TN                     |
| 003-shmem-misc-putnbi-load-retval-local-no.c                     | TN           | TN                     |
| 004-shmem-misc-putnbi-load-memcpy-local-no.c                     | TN           | TN                     |
| 005-shmem-misc-getnbi-load-deep-nesting-local-yes.c              | TP           | TP                     |
| 006-shmem-misc-getnbi-load-aliasing-local-yes.c                  | TP           | TP                     |
| 007-shmem-misc-getnbi-load-retval-local-yes.c                    | TP           | TP                     |
| 008-shmem-misc-getnbi-load-memcpy-local-yes.c                    | FN           | TP                     |
| 009-shmem-misc-get-load-deep-nesting-remote-no.c                 | TN           | TN                     |
| 010-shmem-misc-get-load-funcpointer-remote-no.c                  | TN           | TN                     |
| 011-shmem-misc-get-load-aliasing-remote-no.c                     | TN           | TN                     |
| 012-shmem-misc-get-load-retval-remote-no.c                       | TN           | TN                     |
| 013-shmem-misc-get-load-memcpy-remote-no.c                       | TN           | TN                     |
| 014-shmem-misc-get-store-deep-nesting-remote-yes.c               | TP           | TP                     |
| 015-shmem-misc-get-store-funcpointer-remote-yes.c                | FN           | TP                     |
| 016-shmem-misc-get-store-aliasing-remote-yes.c                   | TP           | TP                     |
| 017-shmem-misc-get-store-retval-remote-yes.c                     | TP           | TP                     |
| 018-shmem-misc-get-store-memcpy-remote-yes.c                     | FN           | TP                     |

## B.3 GASPI Results

|   | RMASanitizer | RMASanitizer-<br>NoOpt |
|---|--------------|------------------------|
| 001-GASPI-conflict-write-load-local-no.c        | TN           | TN                     |
| 002-GASPI-conflict-write-store-local-yes.c      | TP           | TP                     |
| 003-GASPI-conflict-write-write-local-no.c       | TN           | TN                     |
| 004-GASPI-conflict-read-load-local-yes.c        | TP           | TP                     |
| 005-GASPI-conflict-read-store-local-yes.c       | TP           | TP                     |
| 006-GASPI-conflict-read-write-local-yes.c       | TP           | TP                     |
| 007-GASPI-conflict-read-read-local-yes.c        | TP           | TP                     |
| 008-GASPI-conflict-write_list-load-local-no.c   | TN           | TN                     |
| 009-GASPI-conflict-write_list-store-local-yes.c | TP           | TP                     |

## B RMARaceBench Results

|   | RMASanitizer | RMASanitizer-<br>NoOpt |
|---|--------------|------------------------|
| 010-GASPI-conflict-read_list-load-local-yes.c           | TP           | TP                     |
| 011-GASPI-conflict-read_list-store-local-yes.c          | TP           | TP                     |
| 012-GASPI-conflict-write_notify-load-local-no.c         | TN           | TN                     |
| 013-GASPI-conflict-write_notify-store-local-yes.c       | TP           | TP                     |
| 014-GASPI-conflict-read_notify-load-local-yes.c         | TP           | TP                     |
| 015-GASPI-conflict-read_notify-store-local-yes.c        | TP           | TP                     |
| 016-GASPI-conflict-write_list_notify-load-local-no.c    | TN           | TN                     |
| 017-GASPI-conflict-write_list_notify-store-local-yes.c  | TP           | TP                     |
| 018-GASPI-conflict-read_list_notify-load-local-yes.c    | TP           | TP                     |
| 019-GASPI-conflict-read_list_notify-store-local-yes.c   | TP           | TP                     |
| 020-GASPI-conflict-read-load-remote-no.c                | TN           | TN                     |
| 021-GASPI-conflict-read-read-remote-no.c                | TN           | TN                     |
| 022-GASPI-conflict-read-store-remote-yes.c              | TP           | TP                     |
| 023-GASPI-conflict-read-write-remote-yes.c              | TP           | TP                     |
| 024-GASPI-conflict-read-fetchadd-remote-yes.c           | TP           | TP                     |
| 025-GASPI-conflict-write-load-remote-yes.c              | TP           | TP                     |
| 026-GASPI-conflict-write-store-remote-yes.c             | TP           | TP                     |
| 027-GASPI-conflict-write-write-remote-yes.c             | TP           | TP                     |
| 028-GASPI-conflict-write-fetchadd-remote-yes.c          | TP           | TP                     |
| 029-GASPI-conflict-fetchadd-load-remote-yes.c           | TP           | TP                     |
| 030-GASPI-conflict-fetchadd-store-remote-yes.c          | TP           | TP                     |
| 031-GASPI-conflict-fetchadd-fetchadd-remote-no.c        | TN           | TN                     |
| 032-GASPI-conflict-write_list-load-remote-yes.c         | TP           | TP                     |
| 033-GASPI-conflict-write_list-write-remote-yes.c        | TP           | TP                     |
| 034-GASPI-conflict-read_list-load-remote-no.c           | TN           | TN                     |
| 035-GASPI-conflict-read_list-write-remote-yes.c         | TP           | TP                     |
| 036-GASPI-conflict-write_notify-load-remote-yes.c       | TP           | TP                     |
| 037-GASPI-conflict-write_notify-write-remote-yes.c      | TP           | TP                     |
| 038-GASPI-conflict-read_notify-load-remote-no.c         | TN           | TN                     |
| 039-GASPI-conflict-read_notify-write-remote-yes.c       | TP           | TP                     |
| 040-GASPI-conflict-write_list_notify-load-remote-yes.c  | TP           | TP                     |
| 041-GASPI-conflict-write_list_notify-write-remote-yes.c | TP           | TP                     |
| 042-GASPI-conflict-read_list_notify-load-remote-no.c    | TN           | TN                     |
| 043-GASPI-conflict-read_list_notify-write-remote-yes.c  | TP           | TP                     |
| 001-GASPI-sync-wait-local-yes.c                         | TP           | TP                     |
| 002-GASPI-sync-wait-local-no.c                          | TN           | TN                     |
| 003-GASPI-sync-notify-wait-some-local-yes.c             | FN           | FN                     |
| 004-GASPI-sync-notify-wait-some-local-no.c              | TN           | TN                     |
| 005-GASPI-sync-wait-barrier-remote-no.c                 | TN           | TN                     |
| 006-GASPI-sync-wait-barrier-remote-nonconsistent-yes.c  | TP           | TP                     |
| 007-GASPI-sync-wait-barrier-remote-yes.c                | TP           | TP                     |
| 008-GASPI-sync-wait-write-barrier-remote-yes.c          | TP           | TP                     |
| 009-GASPI-sync-notify-wait-some-remote-no.c             | TN           | TN                     |
| 010-GASPI-sync-notify-wait-some-remote-yes.c            | TP           | TP                     |
| 011-GASPI-sync-wait-allreduce-remote-no.c               | TN           | TN                     |
| 012-GASPI-sync-wait-allreduce-remote-yes.c              | TP           | TP                     |
| 013-GASPI-sync-wait-sendrecv-remote-no.c                | TN           | TN                     |
| 014-GASPI-sync-wait-sendrecv-remote-yes.c               | TP           | TP                     |
| 001-GASPI-atomic-fetchadd-fetchadd-remote-no.c          | TN           | TN                     |
| 002-GASPI-atomic-fetchadd-fetchadd-remote-offset-yes.c  | TP           | TP                     |
| 003-GASPI-atomic-fetchadd-fetchadd-remote-offset-no.c   | TN           | TN                     |
| 001-GASPI-hybrid-for-local-yes.c                        | TP           | TP                     |
| 002-GASPI-hybrid-for-ordered-local-no.c                 | TN           | TN                     |
| 003-GASPI-hybrid-master-local-no.c                      | TN           | TN                     |
| 004-GASPI-hybrid-master-local-yes.c                     | TP           | TP                     |
| 005-GASPI-hybrid-single-local-no.c                      | TN           | TN                     |
| 006-GASPI-hybrid-single-local-yes.c                     | TP           | TP                     |
| 007-GASPI-hybrid-section-local-no.c                     | TN           | TN                     |
| 008-GASPI-hybrid-section-local-yes.c                    | TP           | TP                     |
| 009-GASPI-hybrid-task-local-no.c                        | TN           | TN                     |
| 010-GASPI-hybrid-task-local-yes.c                       | TP           | TP                     |
| 011-GASPI-hybrid-for-ordered-remote-no.c                | TN           | TN                     |
| 012-GASPI-hybrid-for-remote-yes.c                       | TP           | TP                     |
| 013-GASPI-hybrid-master-remote-no.c                     | TN           | TN                     |
| 014-GASPI-hybrid-master-remote-yes.c                    | TP           | TP                     |

|   | RMASanitizer | RMASanitizer-<br>NoOpt |
|---|--------------|------------------------|
| 015-GASPI-hybrid-single-remote-no.c                           | TN           | TN                     |
| 016-GASPI-hybrid-single-remote-yes.c                          | TP           | TP                     |
| 017-GASPI-hybrid-section-remote-no.c                          | TN           | TN                     |
| 018-GASPI-hybrid-section-remote-yes.c                         | TP           | TP                     |
| 019-GASPI-hybrid-task-remote-no.c                             | TN           | TN                     |
| 020-GASPI-hybrid-task-remote-yes.c                            | TP           | TP                     |
| 021-GASPI-hybrid-lock-section-barrier-origin-remote-yes.c     | FN           | FN                     |
| 022-GASPI-hybrid-lock-section-barrier-origin-p2p-remote-yes.c | FN           | FN                     |
| 001-GASPI-misc-write-load-deep-nesting-local-no.c             | TN           | TN                     |
| 002-GASPI-misc-write-load-aliasing-local-no.c                 | TN           | TN                     |
| 003-GASPI-misc-write-load-retval-local-no.c                   | TN           | TN                     |
| 004-GASPI-misc-write-load-memcpy-local-no.c                   | TN           | TN                     |
| 005-GASPI-misc-read-load-deep-nesting-local-yes.c             | TP           | TP                     |
| 006-GASPI-misc-read-load-aliasing-local-yes.c                 | TP           | TP                     |
| 007-GASPI-misc-read-load-retval-local-yes.c                   | TP           | TP                     |
| 008-GASPI-misc-read-load-memcpy-local-yes.c                   | TP           | TP                     |
| 009-GASPI-misc-read-load-deep-nesting-remote-no.c             | TN           | TN                     |
| 010-GASPI-misc-read-load-funcpointer-remote-no.c              | TN           | TN                     |
| 011-GASPI-misc-read-load-aliasing-remote-no.c                 | TN           | TN                     |
| 012-GASPI-misc-read-load-retval-remote-no.c                   | TN           | TN                     |
| 013-GASPI-misc-read-load-memcpy-remote-no.c                   | TN           | TN                     |
| 014-GASPI-misc-read-store-deep-nesting-remote-yes.c           | TP           | TP                     |
| 015-GASPI-misc-read-store-funcpointer-remote-yes.c            | FN           | TP                     |
| 016-GASPI-misc-read-store-aliasing-remote-yes.c               | TP           | TP                     |
| 017-GASPI-misc-read-store-retval-remote-yes.c                 | TP           | TP                     |
| 018-GASPI-misc-read-store-memcpy-remote-yes.c                 | FN           | TP                     |



# Bibliography

- [1] Tatsuya Abe, Toshiyuki Maeda, and Mitsuhsa Sato. “Model Checking Stencil Computations Written in a Partitioned Global Address Space Language”. In: *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. May 2013, pages 365–374. DOI: 10.1109/IPDPSW.2013.90.
- [2] Tassadit Célia Aitkaci, Marc Sergent, Emmanuelle Saillard, Denis Barthou, and Guillaume Papauré. “Dynamic Data Race Detection for MPI-RMA Programs”. In: EuroMPI’21. Sept. 2021. URL: <https://hal.science/hal-03374614/>.
- [3] Simone Atzeni, Ganesh Gopalakrishnan, Zvonimir Rakamaric, Dong H. Ahn, Ignacio Laguna, Martin Schulz, Gregory L. Lee, Joachim Protze, and Matthias S. Müller. “ARCHER: Effectively Spotting Data Races in Large OpenMP Applications”. In: *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. May 2016, pages 53–62. DOI: 10.1109/IPDPS.2016.68.
- [4] John Bachan, Scott B. Baden, Steven Hofmeyr, Mathias Jacquelin, Amir Kamil, Dan Bonachea, Paul H. Hargrove, and Hadia Ahmed. “UPC++: A High-Performance Communication Framework for Asynchronous Computation”. In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. May 2019, pages 963–973. DOI: 10.1109/IPDPS.2019.00104.
- [5] Md Abdullah Shahneous Bari, Ujjwal Arora, Varun Hegde, Tony Curtis, and Barbara Chapman. “OpenSHMEM Checker - A Clang Based Static Checker for OpenSHMEM”. In: *2021 20th International Symposium on Parallel and Distributed Computing (ISPDC)*. July 2021, pages 41–48. DOI: 10.1109/ISPDC52870.2021.9521645.
- [6] Brian Barrett, Ronald B. Brightwell, Ryan Grant, Kevin Pedretti, Kyle Wheeler, Keith D. Underwood, Rolf Riesen, Arthur B. Maccabe, Trammel Hudson, and Scott Hemmert. *The Portals 4.1 Network Programming Interface*. SAND2017-3825. Sandia National Lab. (SNL-NM), Albuquerque, NM (United States), Apr. 1, 2017. DOI: 10.2172/1365498.
- [7] OpenMP Architecture Review Board. *OpenMP Application Programming Interface Version 5.2*. Nov. 2021. URL: <https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf> (visited on 2024-12-01).
- [8] Hans-J. Boehm. “How to Miscompile Programs with “Benign” Data Races”. In: *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism*. HotPar’11. USA: USENIX Association, May 26, 2011, page 3.

## Bibliography

- [9] Hans-J. Boehm and Sarita V. Adve. “Foundations of the C++ Concurrency Memory Model”. In: *SIGPLAN Not.* 43.6 (June 7, 2008), pages 68–78. DOI: 10.1145/1379022.1375591.
- [10] Dan Bonachea and Paul H. Hargrove. “GASNet-EX: A High-Performance, Portable Communication Library for Exascale”. In: *Languages and Compilers for Parallel Computing*. Cham: Springer International Publishing, 2019, pages 138–158. DOI: 10.1007/978-3-030-34627-0\_11.
- [11] Greg Bronevetsky and Bronis R. de Supinski. “Complete Formal Specification of the OpenMP Memory Model”. In: *International Journal of Parallel Programming* 35.4 (Aug. 1, 2007), pages 335–392. DOI: 10.1007/s10766-007-0051-4.
- [12] Semih Burak, Ivan R. Ivanov, Jens Domke, and Matthias Müller. “SPMD IR: Unifying SPMD and Multi-value IR Showcased for Static Verification of Collectives”. In: *Recent Advances in the Message Passing Interface*. Cham: Springer Nature Switzerland, 2025, pages 3–20. DOI: 10.1007/978-3-031-73370-3\_1.
- [13] Georgel Calin, Egor Derevenetc, Rupak Majumdar, and Roland Meyer. “A Theory of Partitioned Global Address Spaces”. In: *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2013)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2013. DOI: 10.4230/LIPIcs.FSTTCS.2013.127.
- [14] B.L. Chamberlain, D. Callahan, and H.P. Zima. “Parallel Programmability and the Chapel Language”. In: *The International Journal of High Performance Computing Applications* 21.3 (Aug. 1, 2007), pages 291–312. DOI: 10.1177/1094342007078442.
- [15] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. “X10: An Object-Oriented Approach to Non-Uniform Cluster Computing”. In: *ACM SIGPLAN Notices* 40.10 (Oct. 12, 2005), pages 519–538. DOI: 10.1145/1103845.1094852.
- [16] Zhezhe Chen, James Dinan, Zhen Tang, Pavan Balaji, Hua Zhong, Jun Wei, Tao Huang, and Feng Qin. “MC-Checker: Detecting Memory Consistency Errors in MPI One-Sided Applications”. In: *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’14. New Orleans, LA, USA: IEEE, Nov. 2014, pages 499–510. DOI: 10.1109/SC.2014.46.
- [17] *CLAIx-2018*. URL: <https://hpc.rwth-aachen.de/claix18> (visited on 2024-12-01).
- [18] *CLAIx-2023*. URL: <https://hpc.rwth-aachen.de/claix23> (visited on 2024-12-01).
- [19] OpenSHMEM Committee. *OpenSHMEM Application Programming Interface Version 1.5*. June 8, 2020. URL: [http://openshmem.org/site/sites/default/site\\_files/OpenSHMEM-1.5.pdf](http://openshmem.org/site/sites/default/site_files/OpenSHMEM-1.5.pdf) (visited on 2024-12-01).

- [20] Andrei Marian Dan, Patrick Lam, Torsten Hoefler, and Martin Vechev. “Modeling and Analysis of Remote Memory Access Programming”. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2016 51.10 (Oct. 19, 2016), pages 129–144. DOI: 10.1145/3022671.2984033.
- [21] Mattias De Wael, Stefan Marr, Bruno De Fraine, Tom Van Cutsem, and Wolfgang De Meuter. “Partitioned Global Address Space Languages”. In: *ACM Computing Surveys* 47.4 (July 21, 2015), pages 1–27. DOI: 10.1145/2716320.
- [22] Thanh-Dang Diep, Karl Furlinger, and Nam Thoai. “MC-CChecker: A Clock-Based Approach to Detect Memory Consistency Errors in MPI One-Sided Applications”. In: *Proceedings of the 25th European MPI Users’ Group Meeting*. Barcelona, Spain: ACM, Sept. 23, 2018, pages 1–11. DOI: 10.1145/3236367.3236369.
- [23] Ali Ebnenasir. “UPC-SPIN: A Framework for the Model Checking of UPC Programs”. In: *Proceedings of the 5th International Conference on PGAS Programming Models*. Galveston Island, TX, USA, 2011.
- [24] C. Fidge. “Logical Time in Distributed Computing Systems”. In: *Computer* 24.8 (Aug. 1991), pages 28–33. DOI: 10.1109/2.84874.
- [25] Colin J. Fidge. “Timestamps in Message-Passing Systems That Preserve the Partial Ordering”. In: *Proceedings of the 11th Australian Computer Science Conference* 10 (1987), pages 56–66.
- [26] Cormac Flanagan and Stephen N. Freund. “FastTrack: Efficient and Precise Dynamic Race Detection”. In: *SIGPLAN Not.* 44.6 (June 15, 2009), pages 121–133. DOI: 10.1145/1543135.1542490.
- [27] GASPI Forum. *GASPI: Global Address Space Programming Interface Version 17.1*. Feb. 7, 2017. URL: <https://raw.githubusercontent.com/GASPI-Forum/GASPI-Forum.github.io/master/standards/GASPI-17.1.pdf> (visited on 2024-12-01).
- [28] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.1*. Nov. 2, 2023. URL: <https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf> (visited on 2024-12-01).
- [29] Karl Furlinger, Colin Glass, Jose Gracia, Andreas Knupfer, Jie Tao, Denis Hunich, Kamran Idrees, Matthias Maiterth, Yousri Mhedheb, and Huan Zhou. “DASH: Data Structures and Algorithms with Support for Hierarchical Locality”. In: *EuroPar 2014: Parallel Processing Workshops*. Cham: Springer International Publishing, 2014, pages 542–552. DOI: 10.1007/978-3-319-14313-2\_46.
- [30] D. Geer. “Chip Makers Turn to Multicore Processors”. In: *Computer* 38.5 (May 2005), pages 11–13. DOI: 10.1109/MC.2005.160.
- [31] Tarek El-Ghazawi and Lauren Smith. “UPC: Unified Parallel C”. In: *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. SC ’06. New York, NY, USA: Association for Computing Machinery, Nov. 11, 2006, 27–es. DOI: 10.1145/1188455.1188483.

## Bibliography

- [32] Tarek El-Ghazawi, Vijay Saraswat, and Bradford Chamberlain. *Programming Using the Partitioned Global Address Space (PGAS) Model - Supercomputing Conference Tutorial*. 2008. URL: <https://sc08.supercomputing.org/scyourway/conference/view/tut116.html> (visited on 2024-12-01).
- [33] Sayan Ghosh, Mahantesh Halappanavar, Antonino Tumeo, Ananth Kalyanaraman, and Assefaw H. Gebremedhin. “MiniVite: A Graph Analytics Benchmarking Tool for Massively Parallel Systems”. In: *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. Nov. 2018, pages 51–56. DOI: 10.1109/PMBS.2018.8641631.
- [34] Eric N. Hanson. “The Interval Skip List: A Data Structure for Finding All Intervals That Overlap a Point”. In: *Algorithms and Data Structures*. Berlin, Heidelberg: Springer, 1991, pages 153–164. DOI: 10.1007/BFb0028258.
- [35] Jonas Hartwig. “Extending the SPMD IR by RMA Concepts Showcased for a Static Data Race Detection”. Bachelor’s thesis. RWTH Aachen University, Oct. 25, 2024.
- [36] Tobias Hilbrich, Martin Schulz, Bronis R. de Supinski, and Matthias S. Müller. “MUST: A Scalable Approach to Runtime Error Detection in MPI Programs”. In: *Tools for High Performance Computing 2009*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pages 53–66. DOI: 10.1007/978-3-642-11261-4\_5.
- [37] Tobias Hilbrich, Matthias S. Müller, Bronis R. de Supinski, Martin Schulz, and Wolfgang E. Nagel. “GTI: A Generic Tools Infrastructure for Event-Based Tools in Parallel Systems”. In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. Shanghai, China: IEEE, May 2012, pages 1364–1375. DOI: 10.1109/IPDPS.2012.123.
- [38] Torsten Hoefler, James Dinan, Rajeev Thakur, Brian Barrett, Pavan Balaji, William Gropp, and Keith Underwood. “Remote Memory Access Programming in MPI-3”. In: *ACM Transactions on Parallel Computing 2.2* (July 8, 2015), pages 1–26. DOI: 10.1145/2780584.
- [39] G.J. Holzmann. “The Model Checker SPIN”. In: *IEEE Transactions on Software Engineering 23.5* (May 1997), pages 279–295. DOI: 10.1109/32.588521.
- [40] Chung-Hsing Hsu and Neena Imam. “Assessment of NVSHMEM for High Performance Computing”. In: *International Journal of Networking and Computing 11.1* (2021), pages 78–101. DOI: 10.15803/ijnc.11.1\_78.
- [41] Alexander Hüeck, Tim Jammer, Joachim Jenke, and Christian Bischof. “Investigating the Real-World Applicability of MPI Correctness Benchmarks”. In: *Proceedings of the SC ’23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. SC-W ’23. New York, NY, USA: Association for Computing Machinery, Nov. 12, 2023, pages 230–233. DOI: 10.1145/3624062.3624091.

- [42] Tim Jammer, Alexander Hück, Jan-Patrick Lehr, Joachim Protze, Simon Schwitanski, and Christian Bischof. “Towards a Hybrid MPI Correctness Benchmark Suite”. In: *Proceedings of the 29th European MPI Users’ Group Meeting*. EuroMPI/USA ’22. New York, NY, USA: Association for Computing Machinery, Sept. 14, 2022, pages 46–56. DOI: 10.1145/3555819.3555853.
- [43] Tim Jammer, Simon Schwitanski, Emmanuelle Saillard, Alexander Hück, Joachim Jenke, Radjasouria Vinayagame, and Christian Bischof. “Designing Quality MPI Correctness Benchmarks: Insights and Metrics”. In: *Proceedings of the 2024 IEEE/ACM Eighth International Workshop on Software Correctness for HPC Applications (Correctness)*. Atlanta, GA, USA: IEEE, 2024, pages 222–226. DOI: 10.1109/SCW63240.2024.00034.
- [44] Tim Jammer, Emmanuelle Saillard, Simon Schwitanski, Joachim Jenke, Radjasouria Vinayagame, Alexander Hück, and Christian Bischof. “MPI-BugBench: A Framework for Assessing MPI Correctness Tools”. In: *Recent Advances in the Message Passing Interface*. Cham: Springer Nature Switzerland, 2025, pages 121–137. DOI: 10.1007/978-3-031-73370-3\_8.
- [45] Joachim Jenke, Simon Schwitanski, Isabel Thärigen, and Matthias S. Müller. “Mapping High-Level Concurrency from OpenMP and MPI to ThreadSanitizer Fibers”. In: *Proceedings of the SC ’23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. SC-W ’23. Denver, CO, USA: Association for Computing Machinery, Nov. 12, 2023, pages 187–195. DOI: 10.1145/3624062.3624085.
- [46] Jithin Jose, Sreeram Potluri, Karen Tomko, and Dhabaleswar K. Panda. “Designing Scalable Graph500 Benchmark with Hybrid MPI+OpenSHMEM Programming Models”. In: *Supercomputing*. Berlin, Heidelberg: Springer, 2013, pages 109–124. DOI: 10.1007/978-3-642-38750-0\_9.
- [47] Sem Klauke. “Optimization and Performance Analysis of Data Structures for Race Detection in MPI RMA Programs”. Bachelor’s thesis. RWTH Aachen University, Sept. 29, 2023.
- [48] Roger Kowalewski and Karl Furlinger. “Nasty-MPI: Debugging Synchronization Errors in MPI-3 One-Sided Applications”. In: *Euro-Par 2016: Parallel Processing*. Cham: Springer International Publishing, 2016, pages 51–62. DOI: 10.1007/978-3-319-43659-3\_4.
- [49] Olaf Krzikalla. “Neue Ansätze zur Speicherzugriffsanalyse paralleler Anwendungen mit gemeinsam genutztem Adressraum [New Approaches for Memory Access Analysis of Parallel Applications with a Shared Address Space]”. Technische Universität Dresden, 2018. URL: <https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa2-332004>.

- [50] Olaf Krzikalla, Andreas Knüpfer, Ralph Müller-Pfefferkorn, and Wolfgang E. Nagel. “On the Modelling of One-Sided Communication Systems”. In: 7th International Conference on PGAS Programming Models. The University of Edinburgh, 2013, pages 41–53.
- [51] Olaf Krzikalla, Arne Rempke, and Ralph Müller-Pfefferkorn. “Analyzing One-Sided Communication Using Memory Access Diagrams”. In: *Euro-Par 2023: Parallel Processing Workshops*. Cham: Springer Nature Switzerland, 2024, pages 147–159. DOI: 10.1007/978-3-031-50684-0\_12.
- [52] Ajay D. Kshemkalyani, Min Shen, and Bhargav Voleti. “Prime Clock: Encoded Vector Clock to Characterize Causality in Distributed Systems”. In: *Journal of Parallel and Distributed Computing* 140 (June 1, 2020), pages 37–51. DOI: 10.1016/j.jpdc.2020.02.008.
- [53] Ignacio Laguna, Ryan Marshall, Kathryn Mohror, Martin Ruefenacht, Anthony Skjellum, and Nawrin Sultana. “A Large-Scale Study of MPI Usage in Open-Source HPC Applications”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’19. Denver Colorado: ACM, Nov. 17, 2019, pages 1–14. DOI: 10.1145/3295500.3356176.
- [54] Paul Oliver Lambrich. “Runtime Correctness Checking of MPI I/O in MUST”. Master’s thesis. RWTH Aachen University, 2023. URL: <https://publications.rwth-aachen.de/record/958999>.
- [55] Leslie Lamport. “Time, Clocks, and the Ordering of Events in a Distributed System”. In: *Communications of the ACM* 21.7 (July 1, 1978), pages 558–565. DOI: 10.1145/359545.359563.
- [56] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. “MLIR: Scaling Compiler Infrastructure for Domain Specific Computation”. In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. Feb. 2021, pages 2–14. DOI: 10.1109/CGO51591.2021.9370308.
- [57] Mathieu Laurent, Emmanuelle Saillard, and Martin Quinson. “The MPI Bugs Initiative: A Framework for MPI Verification Tools Evaluation”. In: *2021 IEEE/ACM 5th International Workshop on Software Correctness for HPC Applications (Correctness)*. Nov. 2021, pages 1–9. DOI: 10.1109/Correctness54621.2021.00008.
- [58] Jinpil Lee and Mitsuhsa Sato. “Implementation and Performance Evaluation of XcalableMP: A Parallel Programming Language for Distributed Memory Systems”. In: *2010 39th International Conference on Parallel Processing Workshops*. Sept. 2010, pages 413–420. DOI: 10.1109/ICPPW.2010.62.
- [59] Jan-Patrick Lehr, Tim Jammer, and Christian Bischof. “MPI-CorrBench: Towards an MPI Correctness Benchmark Suite”. In: *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*. HPDC ’21. Virtual Event Sweden: ACM, June 21, 2021, pages 69–80. DOI: 10.1145/3431379.3460652.

- [60] Mingzhe Li, Xiaoyi Lu, Khaled Hamidouche, Jie Zhang, and Dhabaleswar K. Panda. “Mizan-RMA: Accelerating Mizan Graph Processing Framework with MPI RMA”. In: *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*. Dec. 2016, pages 42–51. DOI: 10.1109/HiPC.2016.015.
- [61] Chunhua Liao, Pei-Hung Lin, Joshua Asplund, Markus Schordan, and Ian Karlin. “DataRaceBench: A Benchmark Suite for Systematic Evaluation of Data Race Detection Tools”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC '17*. New York, NY, USA: Association for Computing Machinery, Nov. 12, 2017, pages 1–14. DOI: 10.1145/3126908.3126958.
- [62] Yiqian Liu, Noushin Azami, Corbin Walters, and Martin Burtscher. “The Indigo Program-Verification Microbenchmark Suite of Irregular Parallel Code Patterns”. In: *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. May 2022, pages 24–34. DOI: 10.1109/ISPASS55109.2022.00003.
- [63] Glenn R. Luecke, James Coyle, James Hoekstra, Marina Kraeva, Ying Xu, Mi-Young Park, Elizabeth Kleiman, Olga Weiss, Andre Wehe, and Melissa Yahya. “The Importance of Run-Time Error Detection”. In: *Tools for High Performance Computing 2009*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pages 145–155. DOI: 10.1007/978-3-642-11261-4\_10.
- [64] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. “Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation”. In: *SIGPLAN Not.* 40.6 (June 12, 2005), pages 190–200. DOI: 10.1145/1064978.1065034.
- [65] *Multi-Threaded Executions and Data Races (since C++11)*. URL: <https://en.cppreference.com/w/cpp/language/multithread> (visited on 2024-12-01).
- [66] Nicholas Nethercote and Julian Seward. “Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation”. In: *SIGPLAN Not.* 42.6 (June 10, 2007), pages 89–100. DOI: 10.1145/1273442.1250746.
- [67] Robert H. B. Netzer and Barton P. Miller. “What Are Race Conditions? Some Issues and Formalizations”. In: *ACM Letters on Programming Languages and Systems* 1.1 (Mar. 1, 1992), pages 74–88. DOI: 10.1145/130616.130623.
- [68] Jarek Nieplocha and Bryan Carpenter. “ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-Time Systems”. In: *Parallel and Distributed Processing*. Berlin, Heidelberg: Springer, 1999, pages 533–546. DOI: 10.1007/BFb0097937.
- [69] Robert W. Numrich and John Reid. “Co-Array Fortran for Parallel Programming”. In: *ACM SIGPLAN Fortran Forum* 17.2 (Aug. 1, 1998), pages 1–31. DOI: 10.1145/289918.289920.

## Bibliography

- [70] Robert W. Numrich and John Reid. “Co-Arrays in the next Fortran Standard”. In: *ACM SIGPLAN Fortran Forum* 24.2 (Aug. 1, 2005), pages 4–17. DOI: 10.1145/1080399.1080400.
- [71] NVIDIA. *NVIDIA Collective Communication Library Documentation*. NVIDIA Collective Communication Library. URL: [https://docs.nvidia.com/deeplearning/nccl/archives/nccl\\_2234/user-guide/docs/index.html](https://docs.nvidia.com/deeplearning/nccl/archives/nccl_2234/user-guide/docs/index.html) (visited on 2024-12-01).
- [72] Yussur Mustafa Orazi. “Evaluating Static Analysis Techniques to Accelerate Data Race Detection for MPI RMA”. Bachelor’s thesis. RWTH Aachen University, May 11, 2023. URL: <http://publications.rwth-aachen.de/record/958085>.
- [73] Chang-Seo Park, Koushik Sen, Paul Hargrove, and Costin Iancu. “Efficient Data Race Detection for Distributed Memory Parallel Programs”. In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’11. New York, NY, USA: Association for Computing Machinery, Nov. 12, 2011, pages 1–12. DOI: 10.1145/2063384.2063452.
- [74] Mi-Young Park and Sang-Hwa Chung. “Detecting Race Conditions in One-Sided Communication of MPI Programs”. In: *2009 Eighth IEEE/ACIS International Conference on Computer and Information Science*. June 2009, pages 867–872. DOI: 10.1109/ICIS.2009.170.
- [75] Cornelius Pätzold. “Clock-Based Concurrency Analysis of Hybrid MPI Programs in MUST”. Master’s thesis. RWTH Aachen University, 2024.
- [76] Cornelius Pätzold, Simon Schwitanski, Joachim Jenke, Felix Tomski, and Matthias S. Müller. “Correctness Checking of MPI+OpenMP Applications Using Vector Clocks in MUST”. In: *Proceedings of the 2024 IEEE/ACM Eighth International Workshop on Software Correctness for HPC Applications (Correctness)*. Atlanta, GA, USA: IEEE, 2024, pages 227–231. DOI: 10.1109/SCW63240.2024.00035.
- [77] Eli Pozniansky and Assaf Schuster. “Efficient On-the-Fly Data Race Detection in Multithreaded C++ Programs”. In: *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPOPP ’03. New York, NY, USA: Association for Computing Machinery, June 11, 2003, pages 179–190. DOI: 10.1145/781498.781529.
- [78] Tommaso Pozzetti and Ajay D. Kshemkalyani. “Resettable Encoded Vector Clock for Causality Analysis With an Application to Dynamic Race Detection”. In: *IEEE Transactions on Parallel and Distributed Systems* 32.4 (Apr. 2021), pages 772–785. DOI: 10.1109/TPDS.2020.3032293.
- [79] Joachim Protze. “Modular Techniques and Interfaces for Data Race Detection in Multi-Paradigm Parallel Programming”. Joachim Protze, 2021. URL: <https://publications.rwth-aachen.de/record/824881> (visited on 2024-12-01).

- [80] Xuehai Qian, Koushik Sen, Paul Hargrove, and Costin Iancu. “SReplay: Deterministic Sub-Group Replay for One-Sided Communication”. In: *Proceedings of the 2016 International Conference on Supercomputing*. ICS '16. New York, NY, USA: Association for Computing Machinery, June 1, 2016, pages 1–13. DOI: 10.1145/2925426.2926264.
- [81] Emmanuelle Saillard, Patrick Carribault, and Denis Barthou. “PARCOACH: Combining Static and Dynamic Validation of MPI Collective Communications”. In: *The International Journal of High Performance Computing Applications* 28.4 (Nov. 1, 2014), pages 425–434. DOI: 10.1177/1094342014552204.
- [82] Emmanuelle Saillard, Marc Sergent, Célia Tassadit Ait Kaci, and Denis Barthou. “Static Local Concurrency Errors Detection in MPI-RMA Programs”. In: *2022 IEEE/ACM Sixth International Workshop on Software Correctness for HPC Applications (Correctness)*. Nov. 2022, pages 18–26. DOI: 10.1109/Correctness56720.2022.00008.
- [83] Adrian Schmitz, Joachim Protze, Lechen Yu, Simon Schwitanski, and Matthias S. Müller. “DataRaceOnAccelerator – A Micro-benchmark Suite for Evaluating Correctness Tools Targeting Accelerators”. In: *Euro-Par 2019: Parallel Processing Workshops*. Cham: Springer International Publishing, 2020, pages 245–257. DOI: 10.1007/978-3-030-48340-1\_19.
- [84] Joseph Schuchart, Christoph Niethammer, José Gracia, and George Bosilca. “Quo Vadis MPI RMA? Towards a More Efficient Use of MPI One-Sided Communication”. In: EuroMPI'21. Nov. 15, 2021. URL: <https://arxiv.org/abs/2111.08142v1> (visited on 2024-12-01).
- [85] Martin Schulz and Bronis R. de Supinski. “PNMPI Tools: A Whole Lot Greater than the Sum of Their Parts”. In: *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. SC '07. New York, NY, USA: Association for Computing Machinery, Nov. 10, 2007, pages 1–10. DOI: 10.1145/1362622.1362663.
- [86] Martin Schulz, Greg Bronevetsky, and Bronis R. de Supinski. “On the Performance of Transparent MPI Piggyback Messages”. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Berlin, Heidelberg: Springer, 2008, pages 194–201. DOI: 10.1007/978-3-540-87475-1\_28.
- [87] Reinhard Schwarz and Friedemann Mattern. “Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail”. In: *Distributed Computing* 7.3 (Mar. 1, 1994), pages 149–174. DOI: 10.1007/BF02277859.
- [88] Simon Schwitanski. “On-the-Fly Data Race Detection in MPI One-Sided Communication”. Master’s thesis. RWTH Aachen University, 2017. URL: <https://publications.rwth-aachen.de/record/718908>.

## Bibliography

- [89] Simon Schwitanski, Felix Tomski, Joachim Protze, Christian Terboven, and Matthias S. Müller. “An On-the-Fly Method to Exchange Vector Clocks in Distributed-Memory Programs”. In: *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. Lyon, France: IEEE, May 2022, pages 530–540. DOI: 10.1109/IPDPSW55747.2022.00093.
- [90] Simon Schwitanski, Joachim Jenke, Felix Tomski, Christian Terboven, and Matthias S. Müller. “On-the-Fly Data Race Detection for MPI RMA Programs with MUST”. In: *2022 IEEE/ACM Sixth International Workshop on Software Correctness for HPC Applications (Correctness)*. Dallas, TX, USA: IEEE, Nov. 2022, pages 27–36. DOI: 10.1109/Correctness56720.2022.00009.
- [91] Simon Schwitanski, Joachim Jenke, Sven Klotz, and Matthias S. Müller. “RMARaceBench: A Microbenchmark Suite to Evaluate Race Detection Tools for RMA Programs”. In: *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis. SC-W '23*. Denver, CO, USA: Association for Computing Machinery, Nov. 12, 2023, pages 205–214. DOI: 10.1145/3624062.3624087.
- [92] Simon Schwitanski, Yussur Mustafa Oraj, Cornelius Pätzold, Joachim Jenke, and Matthias S. Müller. “Leveraging Static Analysis to Accelerate Dynamic Race Detection for Remote Memory Access Programs”. In: *ISC High Performance 2024 International Workshops*. Hamburg, Germany: Springer, 2024. DOI: 10.1007/978-3-031-73716-9\_4.
- [93] Simon Schwitanski, Yussur Mustafa Oraj, Cornelius Pätzold, Joachim Jenke, Felix Tomski, and Matthias S. Müller. “RMA Sanitizer: Generalized Runtime Detection of Data Races in Remote Memory Access Applications”. In: *Proceedings of the 53rd International Conference on Parallel Processing. ICPP '24*. Gotland, Sweden: Association for Computing Machinery, Aug. 12, 2024, pages 833–844. DOI: 10.1145/3673038.3673109.
- [94] Konstantin Serebryany and Timur Iskhodzhanov. “ThreadSanitizer: Data Race Detection in Practice”. In: *Proceedings of the Workshop on Binary Instrumentation and Applications. WBIA '09*. New York, NY, USA: Association for Computing Machinery, Dec. 12, 2009, pages 62–71. DOI: 10.1145/1791194.1791203.
- [95] Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov. “Dynamic Race Detection with LLVM Compiler”. In: *Runtime Verification*. Berlin, Heidelberg: Springer, 2012, pages 110–114. DOI: 10.1007/978-3-642-29860-8\_9.
- [96] Min Si, Antonio J. Peña, Jeff Hammond, Pavan Balaji, and Yutaka Ishikawa. “Scaling NWChem with Efficient and Portable Asynchronous Communication in MPI RMA”. In: *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. May 2015, pages 811–816. DOI: 10.1109/CCGrid.2015.48.

- [97] Stephen F. Siegel, Manchun Zheng, Ziqing Luo, Timothy K. Zirkel, Andre V. Marianiello, John G. Edenhofner, Matthew B. Dwyer, and Michael S. Rogers. “CIVL: The Concurrency Intermediate Verification Language”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '15. New York, NY, USA: Association for Computing Machinery, Nov. 15, 2015, pages 1–12. DOI: 10.1145/2807591.2807635.
- [98] Christian Simmendinger, Roman Iakymchuk, Luis Cebamanos, Dana Akhmetova, Valeria Bartsch, Tiberiu Rotaru, Mirko Rahn, Erwin Laure, and Stefano Markidis. “Interoperability Strategies for GASPI and MPI in Large-Scale Scientific Applications”. In: *The International Journal of High Performance Computing Applications* 33.3 (May 2019), pages 554–568. DOI: 10.1177/1094342018808359.
- [99] *SPEC MPI® 2007*. URL: <https://www.spec.org/mpi2007/> (visited on 2024-12-01).
- [100] Monika ten Bruggencate and Duncan Roweth. “DMAPP - An API for One-sided Program Models on Baker Systems”. In: Cray User Group Conference. 2010.
- [101] *ThreadSanitizer Algorithm*. URL: <https://github.com/google/sanitizers/wiki/ThreadSanitizerAlgorithm> (visited on 2024-12-01).
- [102] *ThreadSanitizer v3 Shadow Cell Layout*. URL: [https://github.com/llvm/llvm-project/blob/llvmorg-17-init/compiler-rt/lib/tsan/rtl/tsan\\_shadow.h#L149](https://github.com/llvm/llvm-project/blob/llvmorg-17-init/compiler-rt/lib/tsan/rtl/tsan_shadow.h#L149) (visited on 2024-12-01).
- [103] Felix Tomski. “Developing an On-the-Fly Vector Clock Exchange for Distributed Memory Systems Using the Generic Tool Infrastructure”. Master’s thesis. RWTH Aachen University, 2021. URL: <https://publications.rwth-aachen.de/record/837436>.
- [104] M. Valiev, E. J. Bylaska, N. Govind, K. Kowalski, T. P. Straatsma, H. J. J. Van Dam, D. Wang, J. Nieplocha, E. Apra, T. L. Windus, and W. A. de Jong. “NWChem: A Comprehensive and Scalable Open-Source Solution for Large Scale Molecular Simulations”. In: *Computer Physics Communications* 181.9 (Sept. 1, 2010), pages 1477–1489. DOI: 10.1016/j.cpc.2010.04.018.
- [105] Rob F. Van der Wijngaart and Timothy G. Mattson. “The Parallel Research Kernels”. In: *2014 IEEE High Performance Extreme Computing Conference (HPEC)*. Waltham, MA, USA: IEEE, Sept. 2014, pages 1–6. DOI: 10.1109/HPEC.2014.7040972.
- [106] Radjasouria Vinayagame, Emmanuelle Saillard, Samuel Thibault, Van Man Nguyen, and Marc Sergent. “Rethinking Data Race Detection in MPI-RMA Programs”. In: *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. Denver, CO, USA: ACM, Nov. 12, 2023, pages 196–204. DOI: 10.1145/3624062.3624086.
- [107] Radjasouria Vinayagame, Van Man Nguyen, Marc Sergent, Samuel Thibault, and Emmanuelle Saillard. “Static-Dynamic Analysis for Performance and Accuracy of Data Race Detection in MPI One-Sided Programs”. In: *ISC High Performance 2024 International Workshops*. Hamburg, Germany: Springer, 2024. DOI: 10.1007/978-3-031-73716-9\_5.

## Bibliography

- [108] Anh Vo, Sriram Aananthakrishnan, Ganesh Gopalakrishnan, Bronis R. de Supinski, Martin Schulz, and Greg Bronevetsky. “A Scalable and Distributed Dynamic Formal Verifier for MPI Programs”. In: *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '10. Nov. 2010, pages 1–10. DOI: 10.1109/SC.2010.7.
- [109] Wenwen Wang. “MPIRace: A Static Data Race Detector for MPI Programs”. In: *Languages and Compilers for Parallel Computing*. Cham: Springer Nature Switzerland, 2023, pages 73–90. DOI: 10.1007/978-3-031-31445-2\_6.
- [110] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. “Titanium: A High-Performance Java Dialect”. In: *Concurrency: Practice and Experience* 10.11-13 (1998), pages 825–836. DOI: 10.1002/(SICI)1096-9128(199809/11)10:11/13<825::AID-CPE383>3.0.CO;2-H.
- [111] Qing Zhou, Lian Li, Lei Wang, Jingling Xue, and Xiaobing Feng. “May-Happen-in-Parallel Analysis with Static Vector Clocks”. In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. CGO 2018. New York, NY, USA: Association for Computing Machinery, Feb. 24, 2018, pages 228–240. DOI: 10.1145/3168813.

# List of Figures

|      |  |    |
|------|--|----|
| 2.1  | Memory models used in shared memory, message passing, and PGAS/RMA | 8  |
| 2.2  | Message passing and Remote Memory Access communication             | 10 |
| 2.3  | MPI RMA memory models  | 12 |
| 2.4  | Active target completion modes in RMA                              | 14 |
| 2.5  | Passive target completion modes in RMA                             | 16 |
| 2.6  | Signal and notification mechanisms in RMA                          | 18 |
| 2.7  | Local buffer race examples in MPI RMA, OpenSHMEM, and GASPI        | 20 |
| 2.8  | Remote race examples in MPI RMA using locks                        | 22 |
| 2.9  | Remote race examples using SHMEM barriers                          | 23 |
| 2.10 | Remote race examples using SHMEM wait-until and GASPI wait         | 23 |
| 2.11 | Incorrect atomicity example in MPI RMA                             | 24 |
| 2.12 | Remote race example in MPI RMA and OpenMP                          | 25 |
| 2.13 | Polling example in MPI RMA   | 26 |
|      |  |    |
| 3.1  | Vector clock exchange example with three processes                 | 32 |
| 3.2  | Classification of synchronization in MPI, OpenSHMEM, and GASPI     | 33 |
| 3.3  | Example usage of <i>MPI_Win_lock/unlock</i> in MPI RMA             | 36 |
| 3.4  | Example of handling non-blocking point-to-point operations in MPI  | 41 |
| 3.5  | Collective vector clock exchange example with all-to-all           | 43 |
| 3.6  | Collective vector clock exchange example with all-to-one           | 44 |
| 3.7  | Resource-bound synchronization example                             | 45 |
| 3.8  | Non-blocking receive examples in MPI                               | 47 |
| 3.9  | SHMEM wait examples with ambiguous synchronization behavior        | 48 |
| 3.10 | Tree-based overlay network example in GTI                          | 52 |
| 3.11 | Analysis workflow for the vector clock exchange of two processes   | 54 |
| 3.12 | Polling example in MPI RMA with user annotation functions          | 57 |
| 3.13 | Slowdown factors of the vector clock exchange using tool threads   | 59 |
| 3.14 | Slowdown factors of the vector clock exchange using tool processes | 60 |
| 3.15 | MPI+OpenMP synchronization example with nested vector clocks       | 61 |
| 3.16 | MPI file I/O example   | 63 |
| 3.17 | MPI+OpenMP example with thread-level violation                     | 64 |
| 3.18 | Vector clock representation of the code examples in Figure 3.17    | 65 |
|      |  |    |
| 4.1  | Data race examples in MPI RMA using locks                          | 72 |
| 4.2  | Inference rules of the consistency model                           | 77 |
| 4.3  | Concurrent region examples of local buffer races                   | 82 |

## List of Figures

|      |   |     |
|------|---|-----|
| 4.4  | Remote concurrent region example of Figure 2.8a . . . . .   | 84  |
| 4.5  | Remote concurrent region example using <i>MPI_Win_fence</i> . . . . .                                   | 85  |
| 4.6  | Remote concurrent region example of Figure 2.10a . . . . .  | 86  |
| 4.7  | Identical remote concurrent regions of two $\xrightarrow{fo}$ -ordered <i>shmem_put</i> calls . . . . . | 87  |
| 4.8  | External synchronization of two remote writes . . . . .   | 88  |
| 4.9  | Race checking algorithm for races in RMA . . . . .  | 90  |
|      |   |     |
| 5.1  | Source code instrumentation with TSan . . . . .   | 109 |
| 5.2  | Shadow word storing metadata of a memory access in ThreadSanitizer . . . . .                            | 110 |
| 5.3  | Fiber examples for MPI non-blocking point-to-point communication . . . . .                              | 112 |
| 5.4  | RMASanitizer toolchain workflow . . . . .   | 113 |
| 5.5  | 1D-stencil exchange in MPI RMA using <i>MPI_Put</i> and fences . . . . .                                | 115 |
| 5.6  | Buffer dependence analysis example in LLVM IR . . . . .   | 116 |
| 5.7  | Synchronization and consistency tracking architecture of RMASanitizer . . . . .                         | 119 |
| 5.8  | Exchange of RMA operation information between tool threads . . . . .                                    | 120 |
| 5.9  | Concurrent regions of an <i>MPI_Put</i> annotated as fibers . . . . .                                   | 122 |
| 5.10 | Annotation of a remote concurrent region of an RMA operation in TSan . . . . .                          | 123 |
| 5.11 | RMASanitizer tool workflow . . . . .  | 124 |
| 5.12 | Verification of a race reported by TSan on two memory operations . . . . .                              | 125 |
| 5.13 | Delayed arrival of RMA operation metadata . . . . .   | 127 |
| 5.14 | MPI RMA remote race example with RMASanitizer output . . . . .  | 128 |
| 5.15 | Interval Skip List example . . . . .  | 129 |
|      |   |     |
| 6.1  | RMARaceBench test case example for the conflict category . . . . .                                      | 135 |
| 6.2  | RMARaceBench local buffer race example of the <i>miscellaneous</i> category . . . . .                   | 139 |
| 6.3  | Evaluation results for conflict and synchronization category . . . . .                                  | 143 |
| 6.4  | RMARaceBench remote race example of the <i>synchronization</i> category . . . . .                       | 144 |
| 6.5  | Evaluation results for hybrid and miscellaneous category . . . . .                                      | 145 |
| 6.6  | RMARaceBench SHMEM example with no race . . . . .   | 147 |
| 6.7  | RMARaceBench SHMEM example achieving local completion through notifications . . . . .                   | 148 |
| 6.8  | Overhead results of RMASanitizer on PRK Stencil, LULESH, and miniMD . . . . .                           | 154 |
| 6.9  | Overhead breakdown of the PRK Stencil (MPI RMA) and LULESH run . . . . .                                | 155 |
| 6.10 | Overhead results of RMASanitizer on NPB BT, CFD-Proxy, and miniVite . . . . .                           | 156 |
| 6.11 | Overhead breakdown of NPB BT (MPI RMA) and CFD-Proxy . . . . .  | 157 |
| 6.12 | Overhead results of RMASanitizer and PARCOACH on PRK Stencil, LULESH, and miniVite . . . . .            | 159 |

# List of Tables

|     |   |     |
|-----|---|-----|
| 2.1 | Compatibility of load/store and local buffer operations . . . . .   | 20  |
| 2.2 | Compatibility of RMA operations and local load/store accesses . . . . .                                   | 21  |
| 3.1 | Mapping of point-to-point procedures to signal and wait . . . . .   | 40  |
| 3.2 | Mapping of collective procedures . . . . .  | 42  |
| 3.3 | Mapping of synchronization through locks . . . . .  | 46  |
| 3.4 | Mapping of synchronization through polling . . . . .  | 46  |
| 4.1 | Memory events, sets, and helper functions of the model . . . . .  | 73  |
| 4.2 | Mapping of MPI, SHMEM, and GASPI completion to model completion events . . . . .                          | 76  |
| 6.1 | Number of tests in RMA RaceBench for MPI RMA, SHMEM, and GASPI .  | 136 |
| 6.2 | Classification quality results for the MPI RMA tests . . . . .  | 142 |
| 6.3 | Classification quality metrics for the tools on the different test categories                             | 142 |
| 6.4 | Classification quality results for the GASPI and SHMEM tests of RMA RaceBench for RMA Sanitizer . . . . . | 146 |
| 6.5 | Set of RMA proxy applications were tested with RMA Sanitizer . . . . .                                    | 150 |