

RMARaceBench: A Microbenchmark Suite to Evaluate Race Detection Tools for RMA Programs

Simon Schwitanski

Chair for High Performance Computing, IT Center
RWTH Aachen University
Aachen, Germany
schwitanski@itc.rwth-aachen.de

Sven Klotz

Chair for High Performance Computing, IT Center
RWTH Aachen University
Aachen, Germany
klotz@itc.rwth-aachen.de

Joachim Jenke

Chair for High Performance Computing, IT Center
RWTH Aachen University
Aachen, Germany
jenke@itc.rwth-aachen.de

Matthias S. Müller

Chair for High Performance Computing, IT Center
RWTH Aachen University
Aachen, Germany
mueller@itc.rwth-aachen.de

ABSTRACT

Parallel programming models with Remote Memory Access (RMA), such as MPI RMA, OpenSHMEM, and GASPI, allow processes to modify the memory of other processes directly. Special care is required to avoid concurrent conflicting accesses that lead to data races across processes resulting in undefined behavior. To detect such RMA races, different race detection tools have been developed. However, there is currently no possibility to compare their effectiveness systematically. In this paper, we present RMARaceBench, a microbenchmark suite to evaluate the detection capabilities of RMA race detection tools for MPI RMA, OpenSHMEM, and GASPI. It consists of about 100 synthetic race test cases for each programming model, aiming to cover all possible race scenarios. Using RMARaceBench, we evaluate two MPI RMA race detectors implemented in the correctness tool suites MUST and PARCOACH. The evaluation shows that RMARaceBench can pinpoint the strengths and weaknesses of RMA race detectors.

CCS CONCEPTS

• **Software and its engineering** → **Correctness**; • **Computing methodologies** → **Parallel computing methodologies**.

KEYWORDS

RMA, correctness, data race, MPI, OpenSHMEM, GASPI

ACM Reference Format:

Simon Schwitanski, Joachim Jenke, Sven Klotz, and Matthias S. Müller. 2023. RMARaceBench: A Microbenchmark Suite to Evaluate Race Detection Tools for RMA Programs. In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA*. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3624062.3624087>

SC-W 2023, November 12–17, 2023, Denver, CO, USA

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA*, <https://doi.org/10.1145/3624062.3624087>.

1 INTRODUCTION

In contrast to the classical two-sided message-passing model used for communication in distributed-memory environments, Remote Memory Access (RMA) models provide one-sided communication where processes can directly access the memory of other processes. Especially for algorithms with irregular communication patterns, the asynchronous nature of RMA with fine-grained completion control allows for efficient data exchanges [11]. MPI RMA [20] as well as Partitioned Global Address Space (PGAS) models such as OpenSHMEM [23] and GASPI [6] provide such an RMA model. In recent years, the adoption of RMA to GPUs for efficient intra-kernel communication is also gaining popularity [9].

From the user perspective, the semantic complexity of RMA is significantly higher than in message-passing models: Users must ensure both consistency and synchronization of remote memory accesses through explicit calls. Violating these properties might lead to *data races*. Figure 1 shows such an example for MPI RMA. Data races, typically known from shared-memory programming, are also a severe issue in RMA programs due to their non-deterministic behavior. Accordingly, data race detection tools specifically for RMA models have been developed [1, 27, 29].

Understanding all possible data race situations in RMA models is desirable for (1) users to raise awareness of possible race scenarios and (2) tool developers to consider all those cases in their tools. In this paper, we present *RMARaceBench*, a semantically-driven benchmark suite covering different RMA race scenarios to evaluate RMA race detectors. In particular, we make the following contributions:

- We give an overview of the three RMA programming models, MPI RMA, OpenSHMEM, and GASPI, and showcase semantic similarities and differences.
- We provide a holistic view and classification of races in RMA models, in particular local buffer races and remote races.
- We present a collection of synthetic race test cases as part of a microbenchmark suite covering MPI RMA, OpenSHMEM, and GASPI with high semantic coverage.
- We evaluate two state-of-the-art MPI RMA race detectors implemented in MUST-RMA [29] and PARCOACH [1, 27] with RMARaceBench to showcase its effectiveness in pinpointing strengths and weaknesses of the tools.

```

1 MPI_Barrier(MPI_COMM_WORLD);
2 if (rank == 0) {
3     MPI_Win_lock(1, win);
4     int target = 1; int value;
5     // reads from winbuf[0] at process 1
6     MPI_Get(&value, 1, MPI_INT, target, 0, 1, MPI_INT, win);
7     MPI_Win_unlock(1, win);
8 }
9 if (rank == 1) {
10    winbuf[0] = 42; // conflicting local store at process 1
11 }
12 MPI_Barrier(MPI_COMM_WORLD);

```

Figure 1: Remote race in MPI RMA: Process 0 performs a remote read with the MPI_Get call which conflicts with the local write access at process 1. There is no proper synchronization between the remote and the local access.

The paper is structured as follows: Section 2 gives an overview of RMA programming concepts in MPI RMA, OpenSHMEM, and GASPI. In Section 3, we provide a classification of races in RMA models. Section 4 discusses what information is needed to detect RMA races and which race detection tools already exist. Section 5 describes the microbenchmark suite RMARaceBench. Section 6 presents the classification quality results on the two MPI RMA race detectors MUST-RMA and PARCOACH. In Section 7, we discuss related efforts on classification quality benchmarks and conclude in Section 8.

2 CONCEPTS OF RMA PROGRAMMING

MPI RMA [20], OpenSHMEM [23], and GASPI [6] provide similar mechanisms to exchange data via remote memory access. While MPI RMA is the most generic one defining and supporting various completion and synchronization patterns, OpenSHMEM and GASPI both focus on defining a PGAS (Partitioned Global Address Space) access and memory model. All three programming models follow the SPMD principle and specify C and FORTRAN library interfaces, except OpenSHMEM, which only provides a C interface.

This section highlights similarities and differences by summarizing the most relevant concepts on memory management, communication, atomics, consistency, and synchronization for the three RMA models. Table 1 gives an overview of the essential calls corresponding to the different concepts.

2.1 Memory Management

Prior to any data exchange via RMA, processes have to expose memory regions to make them remotely accessible. For that, processes can allocate an exposed memory region (e.g., MPI_Win_allocate) or expose an existing local memory region (e.g., MPI_Win_create). Those memory management calls are typically called collectively by all processes. GASPI also defines non-collective variants. The remotely accessible memory regions are called *windows* in MPI RMA, *symmetric data* in OpenSHMEM, and *segments* in GASPI. In the following, we will adopt GASPI’s terminology of *segments*.

In MPI RMA and GASPI, the size of segments can be set individually per process, while in OpenSHMEM, the segment size has to be identical on all processes. The segments are identified through opaque handles in MPI RMA and GASPI and through local memory addresses in OpenSHMEM. Using local memory addresses as identifiers allows OpenSHMEM to additionally specify static and global variables as implicit remotely accessible memory locations.

MPI RMA defines two different memory models: In the *separate memory* model, a segment consists of a public and private copy, and coherence between them is established manually through additional calls. The *unified memory* model does not make this distinction and assumes hardware-based coherence. This model comes closest to modern Remote Direct Memory Access (RDMA) architectures as discussed in [8] and is the memory model that OpenSHMEM and GASPI also assume. This paper, therefore, focuses on the unified memory model for MPI RMA.

2.2 Communication

Data access in the RMA models is done through *communication calls* where the invoking process is called *origin* and the remote process is called *target*: *Remote read* calls such as MPI_Get or shmem_get read a specified memory location from the target and write the result back to a specified *local buffer* at the origin. Analogously, *remote write* calls such as MPI_Put or shmem_put write a value supplied via a local buffer to a remote memory location. In addition, there are atomic communication calls updating remote memory locations with atomicity guarantees.

There are subtle differences in the semantics of the calls for the three RMA models: All communication calls in MPI RMA and GASPI are *nonblocking*, i.e., returning from a call does not mean that the

Table 1: RMA Programming Concepts in MPI RMA, OpenSHMEM, and GASPI

Concept	MPI RMA	OpenSHMEM	GASPI
Memory Management	MPI_Win_(allocate create free attach detach)	shmем_(malloc realloc free)	gaspi_segment_(alloc create register free)
Communication	MPI_(Get Put) MPI_(Rget Rput)	shmем_get_(nbi), shmем_put_(nbi) shmем_(p g), shmем_i(put get) shmем_put_signal_(nbi)	gaspi_read, gaspi_write gaspi_read_list, gaspi_write_list gaspi_notify, gaspi_(read write)_notify
Atomics	MPI_(Accumulate Raccumulate) MPI_(Get_accumulate Rget_accumulate) MPI_(Fetch_and_op Compare_and_swap)	shmем_atomic_(set inc add and or xor) shmем_atomic_fetch_(inc add and or xor)_(nbi) shmем_atomic_(fetch swap compare_swap)_(nbi)	gaspi_atomic_fetch_add gaspi_atomic_compare_swap
Consistency	MPI_Win_fence MPI_Win_(post start complete wait) MPI_Win_(lock unlock flush)_(all) MPI_Win_(flush_local flush_local_all)	shmем_barrier_(all) shmем_quiet shmем_fence	gaspi_wait
Synchronization (excerpt)	MPI_Barrier, MPI_Gather, ... MPI_Send, MPI_Recv, ...	shmем_sync_(all), shmем_collect, ... shmем_(signal)_wait_until_(all any _some) shmем_test_(all any _some)	gaspi_barrier, gaspi_allreduce, ... gaspi_notify_waitsome, gaspi_notify_reset

underlying operation is guaranteed to be finished. In OpenSHMEM, most of the communication calls wait for *local completion*, i.e., the calls implicitly block until the provided local buffer can be reused by the application. However, there are also nonblocking variants suffixed with *nbi* such as `shmem_get_nbi` and `shmem_put_nbi`.

2.3 Consistency and Synchronization

Due to the one-sided nature of RMA communication, synchronization is required to avoid unintended side effects such as data races. Since the term “synchronization” is ambiguous when discussing data races in RMA, we explicitly distinguish *consistency*, which characterizes whether the memory effect of an RMA operation is guaranteed to be visible, and (*process*) *synchronization*, which characterizes whether an event in a program execution happened before another event.

Consistency describes whether the memory effect of an RMA operation is visible and distinguishes two completion states: *Local completion* means that the memory access at the origin on the provided local buffer is completed such that the buffer can be safely accessed. *Remote completion* means that the memory effect of an RMA operation is guaranteed to be visible at the target. The RMA models provide different consistency mechanisms that wait for local or remote completion. The variants range from bulk completion of all previously issued RMA operations to fine-grained completion of individual RMA operations. MPI RMA provides three kinds of mechanisms, namely (1) *active target* consistency through collectively invoked *fence* calls, (2) *post-start-complete-wait (PSCW)* as a more fine-grained consistency pattern, and (3) *passive target* consistency using *lock*, *unlock*, and *flush* calls. OpenSHMEM provides bulk-completion calls as the collective `shmem_barrier_all` or the non-collective variant `shmem_quiet`, which ensures remote completion of all previously issued RMA operations on the calling process. GASPI provides a single call `gaspi_wait`, which guarantees local completion but no remote completion.

In addition to consistency, *process synchronization* is required to ensure the correct ordering of RMA operations and local memory accesses. In the case of MPI RMA, the usual synchronization calls provided by MPI, such as barriers or send-receive pairs, can be used. SHMEM and GASPI provide similar constructs. Often, RMA consistency and process synchronization are combined in a single call, e.g., `MPI_Win_fence` guarantees local and remote completion as well as barrier synchronization. SHMEM and GASPI additionally provide a notification mechanism that allows the origin process to explicitly notify the target when an RMA operation is completed.

3 RACES IN RMA MODELS

The correct usage of consistency and synchronization primitives is a key requirement for RMA communication. Incorrect usage leads to data races with undefined behavior. As formalized in [8], a *data race* between two memory operations in MPI RMA is present if (1) the operations are conflicting and (2) they are not ordered by consistency and happened-before order. In [29], we introduced the distinction between *local buffer race* and a *remote race* specifically for MPI RMA. This section discusses how these definitions apply to RMA models in general and discusses races due to incorrect atomicity and hybrid race scenarios due to multithreading.

Table 2: Compatibility Load / Store and local buffer operations to the same memory location

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

3.1 Local Buffer Races

Most of the RMA communication calls are nonblocking and access a *local buffer* at the origin: When reading data from the target, the result is *written* to the local buffer. When writing data to the target, the data to be transmitted is *read* from the local buffer. After an RMA operation is locally completed, the local buffer is safe to be accessed. Table 2 shows a conflict matrix of local memory accesses and buffer accesses to the same memory location: For an RMA operation that triggers a local buffer write, the buffer should not be accessed at all by any other memory operation until the RMA operation is finished. For an RMA operation that triggers a local buffer read, other concurrent reads to the buffer are safe, and only concurrent writes are in conflict. MPI RMA [20, §12.3], OpenSHMEM [23, §4.2], and GASPI [6, §8.2] explicitly state that such concurrent conflicting accesses lead to undefined results. Figure 2 shows an example of a local buffer race in the three RMA models.

3.2 Remote Races

The one-sided nature of RMA requires the user to think carefully about completing RMA operations and synchronizing processes. First, the user has to ensure *at the origin* that the RMA operation is completed remotely, i.e., the memory effects of the operation are visible at the target. Second, at some point after the origin ensures completion, synchronization has to be established between the origin and the target so that the target can also be sure that the RMA operation is completed. Failing to ensure consistency

```

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

```

```

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

```

```

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

```

Figure 2: Local buffer races in MPI RMA, OpenSHMEM, and GASPI: The remote read (get) writes to *localbuf*, while *printf* reads from it before the operation is locally completed.

Table 3: Compatibility of RMA operations to the same memory location at a target process

	Local Load	Local Store	Remote Read	Remote Write	Remote Atomic Read	Remote Atomic Write
Local Load	–	–	✓	✗	✓	✗
Local Store	–	–	✗	✗	✗	✗
Remote Read	✓	✗	✓	✗	✓	✗
Remote Write	✗	✗	✗	✗	✗	✗
Rem. Atomic Read	✓	✗	✓	✗	✓	✓*
Rem. Atomic Write	✗	✗	✗	✗	✓*	✓*

*when adhering to atomicity semantics in the RMA programming model

or synchronization in case of conflicting remote accesses leads to *remote races* with undefined results.

Table 3 shows the conflict matrix of RMA operations to the same memory location at the target that is valid for all three programming models: Either a remote memory access conflicts with a local memory access at the target, as shown in Figure 1, or different remote memory accesses from (multiple) origin processes conflict with each other, as shown in Figure 3. Concurrent *atomic* accesses are not in conflict as long as they adhere to the atomicity semantics in the RMA programming model. The examples shown here only cover a small portion of the error patterns that could occur due to incorrect consistency and synchronization.

3.3 Incorrect Atomicity

When using atomic RMA operations and adhering to the atomicity requirements of the RMA model, concurrent accesses to the same memory location are serialized so that the resulting value at the given memory location is as if the accesses were performed in some serial order. MPI RMA [20, §12.7.1], OpenSHMEM [23, §3.2] and GASPI [6, §10] specify similar requirements on concurrent atomic operations for defined results: (1) The same (basic) data type should be used, and (2) the accesses are aligned correctly and do not overlap due to different byte offsets passed in the RMA communication call. If concurrent atomic RMA operations violate any of those atomicity constraints, the operations are in conflict and thus lead to a remote race. Figure 4 shows an example of a remote race due to different data types used in atomic operations.

```

1 int target_pe = 1;
2 shmем_barrier_all();
3 if (my_pe == 0) {
4     // remote read from symmetric object "remote" at process 1
5     shmем_int_get(&localbuf, &remote, 1, target_pe);
6 }
7 if (my_pe == 2) {
8     localbuf = 2;
9     // remote write to symmetric object "remote" at process 1
10    shmем_int_put(&remote, &localbuf, 1, target_pe);
11 }
12 shmем_barrier_all();

```

Figure 3: Remote race in OpenSHMEM with three processes: Process 0 reads (get) from process 1, while process 2 writes (put) to process 1 without proper synchronization.

```

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 4: Incorrect atomicity in MPI RMA: Process 0 and process 2 perform an atomic update to process 1 with different data types (MPI_SHORT vs. MPI_INT) leading to a remote race.

```

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 sections
13        {
14            #pragma omp section
15            { MPI_Barrier(MPI_COMM_WORLD); } // first thread
16            #pragma omp section
17            { printf("winbuf is %d\n", winbuf[0]); } // second thread
18        } }

```

Figure 5: Remote race in MPI+OpenMP: Process 0 writes to process 1 running a parallel region with two threads and the sections construct; one thread executes an MPI_Barrier for synchronization, the other reads the memory location before the remote write is completed.

3.4 Hybrid Race Scenarios

Combining distributed-memory models such as MPI with shared-memory models such as OpenMP [22] has proven popular in recent years [13]. Therefore, all considered RMA models support multi-threading, i.e., all threads can perform any RMA call at any point in time during execution. This additional level of concurrency poses further challenges: At the origin, the user has to consider that RMA consistency mechanisms ensure completion on process-scope, i.e., an RMA consistency primitive from one thread ensures completion of all previously issued RMA operations by the calling thread *and* all other threads. Missing or incorrect synchronization between the threads might lead to local buffer races. On the target side, the required synchronization with the origin might only be ensured by a single thread, while another thread still accesses the target’s memory location unsynchronized, which leads to a remote race. Figure 5 shows this example for an MPI+OpenMP program. Similar examples can be constructed for OpenSHMEM and GASPI.

4 ANALYSIS TOOL CONSIDERATIONS

Detecting RMA races requires an analysis tool to understand the semantics of the RMA models. For that, the tool has to track (1) RMA communication calls, (2) RMA consistency and synchronization calls, and (3) local memory accesses. While a static analysis tool can extract this information from the source code, a dynamic

analysis tool has to collect it during the execution of the program. In the following, we discuss the required mechanisms to get this information at runtime and give an overview of state-of-the-art tools for RMA race detection and how they use these mechanisms.

4.1 Observing RMA Calls

Since the three considered RMA models are library-based, all their semantics can be attributed to function calls. Thus, an analysis tool can intercept them to get the required information about passed parameters, such as the target process or the addressed remote memory location. The MPI standard defines a profiling interface [20, §15.2] such that each MPI call has a name-shifted PMPI call equivalent. A tool can replace the MPI call with its own version to collect the passed parameters and forward the parameters to the MPI implementation using the name-shifted PMPI call. OpenSHMEM [23, §10] and GASPI [6, §14.2] also define such an interface.

4.2 Observing Local Memory Accesses

Intercepting the RMA calls is not enough for RMA race detection since the interaction with local memory accesses also has to be analyzed. Capturing those local memory accesses requires an analysis tool to instrument the corresponding memory instructions. The instrumentation can either be done at compile time by inserting additional code as done by some analysis tools [1, 29, 30] or by relying on binary instrumentation frameworks [19, 21].

4.3 RMA Race Detection Tools

Different tools address the problem of RMA race detection, particularly for MPI RMA. An early approach proposed in [24] implements a dynamic analysis tool to detect data races between MPI RMA calls. For each allocated remotely accessible memory region, a shadow memory region is allocated to track the status of memory locations on-the-fly. The tool uses the PMPI interface to intercept the MPI RMA calls. It only detects remote races but no local buffer races; in particular, it does not instrument any local memory access.

MC-Checker [4] is a dynamic post-mortem analysis tool that intercepts MPI RMA calls via PMPI and uses compile-time instrumentation to collect information on local memory accesses at runtime. When the execution is finished, it constructs and traverses a DAG representing the consistency and synchronization relations to find RMA races. Since the analysis neglects some transitive synchronization effects for scalability reasons, false alerts might be reported. MC-CChecker [5] tackles this issue by extending MC-Checker with synchronization tracking based on vector clocks.

RMA-Analyzer [1] is a dynamic runtime analysis tool that uses binary search trees to detect RMA races. Similar to MC-Checker, it uses PMPI and compile-time instrumentation to collect information on RMA and local memory accesses. The captured memory access intervals are inserted in a binary search tree and checked for conflicts on the fly. The tool assumes that RMA synchronization calls are invoked collectively, i.e., only applications with bulk-synchronous consistency and synchronization are supported. While this simplifies the detection algorithms, fine-grained consistency and synchronization effects such as lock/unlock or send/receive pairs are not considered and lead to false positives. The same researchers also propose an analysis approach [27] that detects local

buffer races statically: It uses an LLVM pass to traverse the program CFG using BFS searches to find conflicting accesses. Both the dynamic and the static analysis approach have been merged as analysis passes in the correctness checking tool PARCOACH [26].

MUST-RMA [29] is another dynamic runtime analysis tool for RMA race detection. It combines the infrastructure of the MPI correctness checker MUST [7] with the shared-memory race detector ThreadSanitizer [30] to detect RMA races on the fly. Like the other tools, it uses PMPI to intercept the MPI RMA calls to track the status of RMA operations. To detect the synchronization of the MPI processes, MUST-RMA uses vector clocks. The compile-time instrumentation of ThreadSanitizer tracks the local memory accesses. Based on the tracked consistency and synchronization information, MUST-RMA annotates the RMA operations as additional accesses in ThreadSanitizer to detect races at runtime.

For OpenSHMEM, less effort has been spent on race detection: OpenSHMEM-Checker [3] is a static correctness checking tool based on the Clang Static Analyzer framework. It uses symbolic execution to detect different kinds of usage errors. It has experimental support for race detection but only detects races in specific cases. OpenSHMEM-Analyzer [25] is another static correctness-checking tool but does not have support for race detection at all.

For GASPI, a dynamic post-mortem analysis tool [12] uses binary instrumentation to record both GASPI calls and local memory accesses at runtime. Similar to MC-Checker, it builds and traverses a task graph to detect RMA races post-mortem.

5 RMARACEBENCH

In order to systematically evaluate the classification quality of RMA race detection tools, we developed *RMARaceBench*, a semantically-driven microbenchmark suite comprising synthetic race test cases for MPI RMA, OpenSHMEM, and GASPI. The reasoning behind considering all three different models in this work is that their semantics are mostly equivalent such that the same microbenchmark for one RMA model can be translated to the other without much effort. Further, *RMARaceBench* is defined so that it can be easily extended to other RMA models. The methodology is similar to that of existing classification quality benchmarks such as *DataRaceBench* [16] or *MPI-Corrbench* [15] but with a focus on RMA data races. In the following, we will give an overview of our methodology and the developed test case categories.

5.1 Methodology

The main goal of *RMARaceBench* is to have a broad set of microbenchmarks covering the semantics of the different RMA models holistically. We studied all communication, consistency, and synchronization methods provided by the RMA models as discussed in Section 2 and put them in individual test cases. Based on those test cases, RMA race detection tools can be evaluated in terms of classification quality.

Each test case is a self-contained C source code file that can be compiled and executed to ensure compatibility with both static and dynamic tools. We decided to write all test cases in C since all considered RMA models provide a C interface. The test set includes codes with and without RMA races. All test cases are kept simple, i.e., other than the boilerplate code for initialization

Table 4: RMARaceBench test case numbers for the different programming models

Category	# Incorrect / # Correct / # Total Test Cases		
	MPI RMA	SHMEM	GASPI
Conflict	26 / 13 / 39	35 / 14 / 49	32 / 11 / 43
Synchronization	19 / 17 / 36	11 / 11 / 22	7 / 5 / 12
Atomic	7 / 3 / 10	5 / 4 / 9	1 / 2 / 3
Hybrid	12 / 10 / 22	12 / 10 / 22	12 / 10 / 22
Total	64 / 43 / 107	63 / 39 / 102	52 / 28 / 80

and memory management, they just contain a minimal amount of communication and synchronization. Therefore, some test cases seem to be trivial, but in larger codes with complex control flow, such errors might be hard to detect by the user. The test cases are semi-automatically generated based on code templates to ensure maintainability and simple adaptations of parameters.

To have unambiguous evaluation results in terms of classification quality, a test case with a race always includes just one race, where the conflicting accesses can be attributed to exactly two source code lines. Further, the race is always observable because the affected source code lines will be executed. The test cases include meta information such as the affected code lines, the kind of race (local buffer race or remote race), and the kind of conflicting operations.

Based on the race semantics discussed in Section 3, we introduced four different test categories, namely (1) conflict, (2) synchronization, (3) atomic, and (4) hybrid. An overview of the number of test cases for each category is shown in Table 4.

5.2 Conflict Category

An RMA race detector should be able to distinguish conflicting and non-conflicting memory accesses. To systematically evaluate the conflict detection capabilities, we create a test case for each possible combination of concurrent accesses as shown in Table 2 for local buffer accesses and as shown in Table 3 for remote accesses. This leads to 7 pairs of local buffer accesses and 18 pairs of remote accesses; thus, there are 25 possible conflict scenarios. For GASPI, there are only 19 pairs because there is no call providing the semantics of a remote atomic read. For local buffer accesses, we use the code template as shown in Figure 2, iterate through all possible access pairs, and generate a test case for each scenario. For remote accesses, we use code templates similar to Figure 1 and Figure 3.

Some access kinds are represented by multiple RMA communication calls, e.g., `shmem_put` and `shmem_put_nbi` both perform a *remote write*. For the 25 conflict scenarios, we chose one representative for each access kind to avoid an explosion of combinations. Nevertheless, to ensure coverage, we added for the remaining calls a reduced set of test cases. This leads in total to 39, 49, and 43 test cases for MPI RMA, OpenSHMEM, and GASPI, respectively.

5.3 Synchronization Category

Besides conflict detection in case of concurrent accesses, an RMA race detector has to understand when the memory effect of an RMA communication call is visible and guaranteed to be finished. For

that, it needs to track the consistency and synchronization provided by the RMA models. The *synchronization* category considers pairs of conflicting memory accesses and tests through the different consistency and synchronization mechanisms as shown in Table 1: For each mechanism, there is at least one test case with a race where the calls are missing or inappropriately used and at least one test case without a race where the mechanism is used correctly. This enables RMARaceBench to precisely classify which consistency and synchronization mechanism a tool supports and which it does not.

MPI RMA has the highest number of different consistency mechanisms (fences, PSCW, lock/unlock, flush, flush-local), which are all covered and, therefore, also has the highest number of 36 test cases, followed by OpenSHMEM with 22 test cases. Since the GASPI model is comparably simple regarding consistency and synchronization calls, RMARaceBench provides only 12 test cases.

5.4 Atomic Category

Concurrent access to the same remote memory location with atomic accesses is defined behavior as long as the atomicity requirements of the RMA model are met. As discussed in Section 3, using different access data types or different alignments, the atomicity of the accesses is not guaranteed which might lead to data races. The *atomic* category checks whether the RMA race detector can correctly identify whether atomicity guarantees are given or not. As for the other categories, it includes test cases where the atomicity requirements are met and other test cases where one of the requirements is not fulfilled, leading to a remote race.

5.5 Hybrid Category

In the *hybrid* category, the RMA models are combined with OpenMP as representative of a shared-memory programming model, leading to more complex race scenarios. In the local buffer race test cases, for example, one thread performs the local buffer access while another thread concurrently accesses the same memory location in a conflicting way. In the remote race test cases, for example, one thread synchronizes correctly with the origin process, but another thread accesses the remote memory location concurrently, as shown in Figure 5.

The test cases cover the most important concepts of OpenMP, in particular, the single and master construct, tasks, sections, and work-sharing loops, partly in conjunction with OpenMP barriers to ensure synchronization. For each of the mentioned concepts, we generated at least one test case without race and at least one test case with race due to missing RMA consistency resulting from missing OpenMP synchronization. This leads, in total, to 10 local buffer access test cases and 12 remote access test cases for each RMA model. The test cases have been designed so that a tool has to capture the OpenMP parallelism correctly to detect the races.

6 EVALUATION

To evaluate RMARaceBench, we first executed all test cases and checked whether the RMA model implementations themselves could pinpoint some errors and second used the MPI RMA race test cases to compare the publicly available tools MUST-RMA [29] and PARCOACH [1, 26, 27] in terms of classification quality.

Table 5: Classification quality results of MUST-RMA and PARCOACH (dynamic and static analysis) for the different categories

	MUST-RMA								PARCOACH-dynamic								PARCOACH-static							
	TP	FP	TN	FN	TO	P	R	A	TP	FP	TN	FN	TO	P	R	A	TP	FP	TN	FN	TO	P	R	A
Conflict	13	0	13	13	0	1.00	0.50	0.67	10	1	12	16*	0	0.91	0.38	0.56	5	0	3	7	0	1.00	0.42	0.53
Sync	19	1	16	0	0	0.95	1.00	0.97	4	4	5	7*	16	0.50	0.36	0.45	5	3	3	1	0	0.62	0.83	0.67
Atomic	0	0	4	6	0	-	0.00	0.40	1	4	0	5	0	0.20	0.17	0.10	0	0	0	0	0	-	-	-
Hybrid	10	0	10	2	0	1.00	0.83	0.91	4	5	0	5	8	0.44	0.44	0.29	4	5	0	1	0	0.44	0.80	0.40
Total	42	1	43	21	0	0.98	0.67	0.79	19	14	17	33	24	0.58	0.37	0.43	14	8	6	9	0	0.64	0.61	0.54

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

*Some FNs in PARCOACH-dynamic are systematic due to falsely detected local buffer races preventing detection of the actual remote race, see Section 6.3 for details.

6.1 Setup

All the test cases and the execution environment are available at <https://github.com/RWTH-HPC/RMARaceBench>. The tests were executed in a Docker environment based on a Debian 12 image that uses Clang 15 as the compiler, OpenMPI 4.1.4 for the MPI RMA tests, Sandia OpenSHMEM 1.5.1 for the SHMEM tests, and GPI-2 1.5.1 for the GASPI tests. Most of the tests run with two processes; the maximum number of required processes is four. The hybrid test cases use two processes with up to two OpenMP threads per process. For MUST-RMA, we used version 1.9.0-rma¹, and for PARCOACH, we used version 2.3.1².

6.2 Test Case Runtime Behavior

We ran all test cases to evaluate whether the RMA model implementations themselves report some faulty codes. For all test cases, the execution terminated without any message that there is a race or that the program behaves faulty. Still, the test cases containing races show non-deterministic behavior depending on the instruction interleaving. This missing feedback to the user confirms that data races have to be debugged manually or with tool support.

6.3 MUST-RMA and PARCOACH Comparison

We compared the two publicly available state-of-the-art RMA race detectors, MUST-RMA and PARCOACH, using RMARaceBench. Each test case was executed once, and the results were classified according to the following categories:

- True Positive (TP): Correct report in a test case with a race.
- False Positive (FP): False report in a race-free test case.
- True Negative (TN): No race report in a race-free test case.
- False Negative (FN): No race report in a test case with a race.
- Timeout (TO): Execution did not finish within 30 seconds.

For a true positive (TP), we require the source code lines of the conflicting accesses to be reported. Otherwise, if only a race on an unrelated call or variable access is reported instead, it counts as a false negative (FN). A timeout limit of 30 seconds was chosen because the test cases are small and execute within less than a second without a tool. Based on the results, we calculated per category the precision $P = \frac{TP}{TP+FP}$, the recall $R = \frac{TP}{TP+FN}$ and the accuracy $A = \frac{TP+TN}{TP+FP+TN+FN}$.

¹<https://hpc.rwth-aachen.de/must/files/MUST-v1.9.0-rma.tar.gz>

²<https://gitlab.inria.fr/parcoach/parcoach/-/tree/2.3.1>

Since PARCOACH implements a dynamic analysis pass [1] to detect both local buffer races and remote races and a static analysis pass [27] only for local buffer races, we treat them as separate tools in the evaluation as *PARCOACH-dynamic* and *PARCOACH-static*, respectively. Further, since *PARCOACH-static* only supports the detection of local buffer races, we only evaluated it on the test cases that focus on local buffer races. The results of the tools in the different test categories are shown in Table 5. For detailed results for each test case, we refer to Section A.3 in the appendix.

In the *conflict* category, all three tools show a good precision P but have a high amount of false negatives (FN), leading to a low recall R . The main reason for that is that all tools only analyze the MPI RMA calls *Get*, *Put*, and *Accumulate*, but not the advanced atomic calls such as *Compare_and_swap* or *Fetch_and_op* which are additionally covered in the test cases. Three of the false negatives (FN) with PARCOACH-dynamic are due to wrong reported data race locations: In some remote race tests, the tool falsely detects local buffer races on source code lines that do not have a race, because it sometimes misinterprets the order of local memory accesses and local buffer accesses. When the tool detects a race, it immediately stops the execution with `MPI_Abort`, i.e., even if the remote race could be detected, the false detection of the local buffer race would prevent the report due to the aborted execution. Similarly, the single false positive (FP) is due to a falsely reported local buffer race.

In the *sync* category, MUST-RMA shows a high accuracy A since it understands all kinds of MPI RMA synchronization correctly. There is one false positive (FP) since MUST-RMA may miss synchronization from remote processes in certain cases, as discussed in [29]. PARCOACH-dynamic shows a higher number of timeouts because it deadlocks whenever *Win_lock/unlock* is used, those cases are not counted for the precision, recall, and accuracy. Further, as discussed in Section 4, PARCOACH assumes that all RMA consistency and synchronization calls are executed collectively, i.e., any synchronization other than *fence* or collective *lock-all* calls is not detected. Thus, the number false positives and negatives is higher. The same is true for PARCOACH-static, which does not understand all synchronization and further—due to its static nature—does not understand branching based on the process number leading to false positives. Moreover, for PARCOACH-dynamic, four of the false negatives (FN) and two of the false positives (FP) are again due to falsely reported local buffer races as previously described.

MUST-RMA and PARCOACH-dynamic both have a low accuracy for the *atomic* category because they do not support detecting such

races. MUST-RMA does not report any (correct or false) race at all. Since the *atomic* category only contains tests with remote races, there were no tests executed with PARCOACH-static. For the *hybrid* category, the results are mixed: MUST-RMA does not support thread synchronization but profits from relying on ThreadSanitizer, which includes Archer [2] that captures OpenMP synchronization. However, since MUST-RMA considers any MPI synchronization as global for all threads, false negatives are possible due to this conservative assumption. Both PARCOACH analyses do not support OpenMP, accordingly, the accuracy is lower than for MUST-RMA.

In summary, MUST-RMA shows the higher total accuracy of 0.79 with very high precision of 0.98 due to only one false positive out of the total 107 test cases. PARCOACH-dynamic and PARCOACH-static both show a lower accuracy of 0.43 and 0.54, respectively, since they currently only support a limited number of MPI RMA consistency and synchronization methods. For PARCOACH-dynamic, there are also sometimes misinterpreted orderings of local memory accesses and RMA operations leading to wrong classifications. The evaluation shows that RMA RaceBench successfully pinpoints the strengths and weaknesses of the RMA race detection tools.

7 RELATED WORK

There have been different efforts on classification quality benchmarks for correctness-checking tools. Many of those efforts focus on data race detection: DataRaceBench [16] provides a microbenchmark suite consisting of OpenMP test cases with and without race. Based on initially 140 but now 200 test cases, the authors evaluated different OpenMP race detectors regarding classification quality. DataRaceOnAccelerator [28] adapts the idea of DataRaceBench to offloading programming with accelerators using OpenMP, OpenACC, and CUDA. It consists of roughly 40 test cases per programming model and covers data races between the host and the accelerator and on the accelerator itself. The Indigo Microbenchmark [17] suite focuses on OpenMP and CUDA race test cases that implement certain code patterns of parallel graph applications. The 1720 test cases are automatically generated out of different mutations of a few code templates.

For the verification of MPI applications, two recent efforts are MPI-Corrbench [15] and the MPI Bugs Initiative [14]. Both implement microbenchmarks for various error patterns in MPI, such as erroneous arguments, message races, deadlocks, and resource leaks and test them on different MPI verification tools. MPI-Corrbench consists of 510 synthetic microbenchmarks and provides some mini-apps with injected errors. It also considers argument errors and misplaced calls in MPI RMA but does not contain any RMA race test cases. MPI-Corrbench has been extended to hybrid models for MPI+OpenMP programs [10]; the new test cases, in particular, cover races with OpenMP in the context of MPI nonblocking point-to-point communication. The MPI Bugs Initiative (MBI) [14] focuses on automatically generated test cases out of code templates. They generate a total of 1700 tests, where 45 consider races in MPI RMA (both local buffer and remote races). Those test cases, however, cover only a reduced set of MPI RMA synchronization calls (fence, lock/unlock, lock-all/unlock-all) compared to RMA RaceBench, which for example, also covers flushes, PSCW, interaction with barrier and send/recv synchronization, and also provides equivalent test cases

for OpenSHMEM and GASPI. Moreover, MBI does not cover races related to RMA atomics and the interaction with hybrid parallelism.

The Run-Time Error Detection suite [18] provides about 10,000 error test cases for MPI, OpenMP, and UPC [31] programs. It also contains 86 MPI RMA test cases with different RMA races but no correct codes making it hard to use for classification quality studies. Further, the tests only cover the MPI-2.0 standard, which misses some synchronization mechanisms (flushes, lock-all) and especially the updated unified memory model introduced in MPI-3.0.

To the best of our knowledge, there is no microbenchmark suite for verification tools that covers OpenSHMEM or GASPI at all. Thus, RMA RaceBench is the first approach that (1) focuses explicitly on the semantic coverage of RMA races and (2) covers the three RMA models MPI RMA, OpenSHMEM, and GASPI.

8 CONCLUSION

RMA programming models allow processes to access the memory of other processes directly but require the user to ensure consistency and synchronization to avoid data races. This paper considered MPI RMA, OpenSHMEM, and GASPI as programming models and evaluated different race scenarios. We classified RMA races into local buffer races, remote races, races due to incorrect atomicity, and races in hybrid scenarios with OpenMP. Due to the semantic similarity of MPI RMA, OpenSHMEM, and GASPI, this classification holds for all three models and can be extended to other RMA programming models.

Based on the classification and the different race scenarios, we created RMA RaceBench. This semantically-driven microbenchmark suite contains about 100 race test cases for each MPI RMA, OpenSHMEM, and GASPI and can be used to compare current and future generations of RMA race detectors. The test cases are grouped into the different categories *conflict*, *synchronization*, *atomic*, and *hybrid*, which systemically test through different kinds of conflicting accesses, consistency and synchronization methods, atomic operations, and hybrid synchronization scenarios, respectively.

Running the race test cases without any verification tool showed that none of the RMA implementations provides any measure to detect and report races, so it is up to the user to spot them. The evaluation of the race detectors implemented in MUST-RMA and PARCOACH shows that races of the *conflict* and *synchronization* category can mainly be detected. However, for conflicting atomic operations or race scenarios where the interaction with OpenMP has to be considered, the support for some cases is currently limited.

For future work, the test cases in RMA RaceBench could be extended by a mechanism to automatically generate mutations that systematically test out different input sizes or different data types to get an even higher coverage. Another aspect could be extending real-world programs with injected data races to check whether the tools also detect them.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the German Federal Ministry of Education and Research (BMBF) and the state of North Rhine-Westphalia for supporting this work as part of the NHR funding.

REFERENCES

- [1] Tassadit Célia Aitkaci, Marc Sergent, Emmanuelle Saillard, Denis Barthou, and Guillaume Papauré. 2021. Dynamic Data Race Detection for MPI-RMA Programs. In *EuroMPI '21 - European MPI Users' Group Meeting*.
- [2] Simone Atzeni, Ganesh Gopalakrishnan, Zvonimir Rakamaric, Dong H. Ahn, Ignacio Laguna, Martin Schulz, Gregory L. Lee, Joachim Protze, and Matthias S. Müller. 2016. ARCHER: Effectively Spotting Data Races in Large OpenMP Applications. In *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*. IEEE, 53–62. <https://doi.org/10.1109/IPDPS.2016.68>
- [3] Md Abdullah Shahneous Bari, Ujjwal Arora, Varun Hegde, Tony Curtis, and Barbara M. Chapman. 2021. OpenSHMEM Checker - A Clang Based Static Checker for OpenSHMEM. In *20th International Symposium on Parallel and Distributed Computing, ISPDC 2021, Cluj-Napoca, Romania, July 28-30, 2021*. IEEE, 41–48. <https://doi.org/10.1109/ISPDC52870.2021.9521645>
- [4] Zhezhe Chen, James Dinan, Zhen Tang, Pavan Balaji, Hua Zhong, Jun Wei, Tao Huang, and Feng Qin. 2014. MC-Checker: Detecting Memory Consistency Errors in MPI One-Sided Applications. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16-21, 2014*. IEEE, 499–510. <https://doi.org/10.1109/SC.2014.46>
- [5] Thanh-Dang Diep, Karl Furlinger, and Nam Thoai. 2018. MC-CChecker: A Clock-Based Approach to Detect Memory Consistency Errors in MPI One-Sided Applications. In *EuroMPI'18: European MPI Users' Group Meeting, Barcelona, Spain, September 23-26, 2018*. ACM, 9:1–9:11. <https://doi.org/10.1145/3236367.3236369>
- [6] GASPI Forum. 2017. GASPI: Global Address Space Programming Interface 17.1. <https://raw.githubusercontent.com/GASPI-Forum/GASPI-Forum.github.io/master/standards/GASPI-17.1.pdf> [online; accessed 26-September-2023].
- [7] Tobias Hilbrich, Martin Schulz, Bronis R. de Supinski, and Matthias S. Müller. 2009. MUST: A Scalable Approach to Runtime Error Detection in MPI Programs. In *Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing, September 2009, ZIH, Dresden*. Springer, 53–66. https://doi.org/10.1007/978-3-642-11261-4_5
- [8] Torsten Hoefler, James Dinan, Rajeev Thakur, Brian Barrett, Pavan Balaji, William Gropp, and Keith D. Underwood. 2015. Remote Memory Access Programming in MPI-3. *ACM Transactions on Parallel Computing* 2, 2 (2015), 9:1–9:26. <https://doi.org/10.1145/2780584>
- [9] Chung-Hsing Hsu, Neena Imam, Akhil Langer, Sreeram Potluri, and Chris J. Newburn. 2020. An Initial Assessment of NVSHMEM for High Performance Computing. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2020, New Orleans, LA, USA, May 18-22, 2020*. IEEE, 617–626. <https://doi.org/10.1109/IPDPSW50202.2020.00104>
- [10] Tim Jammer, Alexander Hück, Jan-Patrick Lehr, Joachim Protze, Simon Schwitanski, and Christian H. Bischof. 2022. Towards a Hybrid MPI Correctness Benchmark Suite. In *EuroMPI/USA'22: European MPI Users' Group Meeting, Chattanooga, TN, USA, September 26-28, 2022*. ACM, 46–56. <https://doi.org/10.1145/3555819.3555853>
- [11] Jithin Jose, Sreeram Potluri, Karen Tomko, and Dhableswar K. Panda. 2013. Designing Scalable Graph500 Benchmark with Hybrid MPI+OpenSHMEM Programming Models. In *Supercomputing - 28th International Supercomputing Conference, ISC 2013, Leipzig, Germany, June 16-20, 2013. Proceedings*, Vol. 7905. Springer, 109–124. https://doi.org/10.1007/978-3-642-38750-0_9
- [12] Olaf Krzikalla. 2018. *Neue Ansätze zur Speicherzugriffsanalyse paralleler Anwendungen mit gemeinsam genutztem Adressraum [New approaches for memory access analysis of parallel applications with a shared address space]*. Dissertation. Technische Universität Dresden. <https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa2-332004>
- [13] Ignacio Laguna, Ryan J. Marshall, Kathryn M. Mohror, Martin Ruefenacht, Anthony Skjellum, and Nawrin Sultana. 2019. 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 2019, Denver, Colorado, USA, November 17-19, 2019*. ACM, 31:1–31:14. <https://doi.org/10.1145/3295500.3356176>
- [14] Mathieu Laurent, Emmanuelle Saillard, and Martin Quinson. 2021. The MPI Bugs Initiative: a Framework for MPI Verification Tools Evaluation. In *5th IEEE/ACM International Workshop on Software Correctness for HPC Applications, Correctness@SC 2021, St. Louis, MO, USA, November 19, 2021*. IEEE, 1–9. <https://doi.org/10.1109/Correctness54621.2021.00008>
- [15] Jan-Patrick Lehr, Tim Jammer, and Christian H. Bischof. 2021. MPI-CorrBench: Towards a MPI Correctness Benchmark Suite. In *HPDC '21: The 30th International Symposium on High-Performance Parallel and Distributed Computing, Virtual Event, Sweden, June 21-25, 2021*. ACM, 69–80. <https://doi.org/10.1145/3431379.3460652>
- [16] Chunhua Liao, Pei-Hung Lin, Joshua Asplund, Markus Schordan, and Ian Karlin. 2017. 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 2017, Denver, CO, USA, November 12 - 17, 2017*. ACM, 11. <https://doi.org/10.1145/3126908.3126958>
- [17] Yiqian Liu, Noushin Azami, Corbin Walters, and Martin Burtscher. 2022. The Indigo Program-Verification Microbenchmark Suite of Irregular Parallel Code Patterns. In *International IEEE Symposium on Performance Analysis of Systems and Software, ISPASS 2022, Singapore, May 22-24, 2022*. IEEE, 24–34. <https://doi.org/10.1109/ISPASS55109.2022.00003>
- [18] Glenn R. Luecke, James Coyle, James Hoekstra, Marina Kraeva, Ying Xu, Mi-Young Park, Elizabeth Kleiman, Olga Weiss, André Wehe, and Melissa Yahya. 2009. The Importance of Run-Time Error Detection. In *Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing, September 2009, ZIH, Dresden*. Springer, 145–155. https://doi.org/10.1007/978-3-642-11261-4_10
- [19] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*. ACM, 190–200. <https://doi.org/10.1145/1065010.1065034>
- [20] Message Passing Interface Forum. 2021. MPI: A Message-Passing Interface Standard Version 4.0. <http://mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf> [online; accessed 26-September-2023].
- [21] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavy-weight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*. ACM, 89–100. <https://doi.org/10.1145/1250734.1250746>
- [22] OpenMP Architecture Review Board. 2021. OpenMP Application Programming Interface Version 5.2. <https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf> [online; accessed 26-September-2023].
- [23] OpenSHMEM Committee. 2020. OpenSHMEM: Application Programming Interface Version 1.5. http://openshmem.org/site/sites/default/site_files/OpenSHMEM-1.5.pdf [online; accessed 26-September-2023].
- [24] Mi-Young Park and Sang-Hwa Chung. 2009. Detecting Race Conditions in One-Sided Communication of MPI Programs. In *8th IEEE/ACIS International Conference on Computer and Information Science, IEEE/ACIS ICIS 2009, June 1-3, 2009, Shanghai, China*. IEEE, 867–872. <https://doi.org/10.1109/ICIS.2009.170>
- [25] Swaroop Pophale, Oscar R. Hernandez, Stephen W. Poole, and Barbara M. Chapman. 2014. Extending the OpenSHMEM Analyzer to Perform Synchronization and Multi-valued Analysis. In *OpenSHMEM and Related Technologies. Experiences, Implementations, and Tools - First Workshop, OpenSHMEM 2014, Annapolis, MD, USA, March 4-6, 2014. Proceedings*, Vol. 8356. Springer, 134–148. https://doi.org/10.1007/978-3-319-05215-1_10
- [26] Emmanuelle Saillard, Patrick Carribault, and Denis Barthou. 2014. PARCOACH: Combining static and dynamic validation of MPI collective communications. *Int. J. High Perform. Comput. Appl.* 28, 4 (2014), 425–434. <https://doi.org/10.1177/1094342014552204>
- [27] Emmanuelle Saillard, Marc Sergent, Célia Tassadit Ait Kaci, and Denis Barthou. 2022. Static Local Concurrency Errors Detection in MPI-RMA Programs. In *Sixth IEEE/ACM International Workshop on Software Correctness for HPC Applications, Correctness@SC 2022, Dallas, TX, USA, November 13-18, 2022*. IEEE, 18–26. <https://doi.org/10.1109/Correctness56720.2022.00008>
- [28] Adrian Schmitz, Joachim Protze, Lechen Yu, Simon Schwitanski, and Matthias S. Müller. 2019. DataRaceOnAccelerator - A Micro-benchmark Suite for Evaluating Correctness Tools Targeting Accelerators. In *Euro-Par 2019: Parallel Processing Workshops - Euro-Par 2019 International Workshops, Göttingen, Germany, August 26-30, 2019, Revised Selected Papers*, Vol. 11997. Springer, 245–257. https://doi.org/10.1007/978-3-030-48340-1_19
- [29] Simon Schwitanski, Joachim Jenke, Felix Tomski, Christian Terboven, and Matthias S. Müller. 2022. On-the-Fly Data Race Detection for MPI RMA Programs with MUST. In *Sixth IEEE/ACM International Workshop on Software Correctness for HPC Applications, Correctness@SC 2022, Dallas, TX, USA, November 13-18, 2022*. IEEE, 27–36. <https://doi.org/10.1109/Correctness56720.2022.00009>
- [30] Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov. 2011. Dynamic Race Detection with LLVM Compiler. In *Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers*, Vol. 7186. Springer, 110–114. https://doi.org/10.1007/978-3-642-29860-8_9
- [31] UPC Consortium. 2017. UPC Language Specifications Version 1.3. <https://upc.lbl.gov/publications/upc-spec-1.3.pdf> [online; accessed 26-September-2023].

A ARTIFACT DESCRIPTION

This artifact description explains how to reproduce the evaluation results. The test cases and all required tooling is available at <https://github.com/RWTH-HPC/RMARaceBench>. Our results are available at <https://github.com/RWTH-HPC/RMARaceBench-Results> and at <https://doi.org/10.5281/zenodo.8377982>.

A.1 Software Requirements

The following software packages were used for the evaluation in a Debian 12 Docker environment:

- LLVM / Clang 15.0.6, CMake 3.25.1, Python 3
- OpenMPI 4.1.4, Sandia OpenSHMEM 1.5.1, GPI 1.5.1
- MUST-RMA 1.9.0, PARCOACH 2.3.1

A.2 Reproduce Experiments

We provide a Dockerfile with the required software environment to reproduce our results. In the following, we assume that \$ROOT is the root folder of the supplemental repository / unpacked files. First, build the Docker image with tag rmaracebench and start a shell within the container:

```
# cd $ROOT
# docker build . -t rmaracebench
# docker run -it rmaracebench bash
```

Within the Docker image, use run_test.py to run the plain tests on MPI RMA, SHMEM, GASPI without any tool and specify an output folder to store the results:

```
# python run_test.py plain -o results-plain
```

The same script run_test.py can be used to execute the classification quality tests on MUST-RMA and PARCOACH:

```
# python run_test.py tools -o results-tools
```

In both cases, the result folder contains the outputs of the different test cases runs for further investigations. For the classification quality tests, a CSV file is also generated that can be used with the Python script parse_results.py to generate the result tables:

```
# python parse_results.py result_folder/results.csv
```

A.3 RMA Race Bench Results

Table 6 and Table 7 show the results of MUST-RMA, PARCOACH-dynamic, and PARCOACH-static on the MPI RMA race test cases.

Table 6: Results for Conflict category

Test Name	MUST-RMA	PARCOACH-dynamic	PARCOACH-static
001-MPI-conflict-put-load-local-no.c	TN	TN	TN
002-MPI-conflict-put-store-local-yes.c	TP	TP	TP
003-MPI-conflict-put-put-local-no.c	TN	TN	TN
004-MPI-conflict-get-load-local-yes.c	TP	TP	TP
005-MPI-conflict-get-store-local-yes.c	TP	TP	TP
006-MPI-conflict-get-put-local-yes.c	TP	TP	TP
007-MPI-conflict-get-get-local-yes.c	TP	TP	TP
008-MPI-conflict-acc-store-local-yes.c	TP	TP	FN
009-MPI-conflict-acc-load-local-no.c	TN	TN	TN
010-MPI-conflict-gacc-store-local-yes.c	TP	FN	FN
011-MPI-conflict-gacc-load-local-yes.c	TP	FN	FN
012-MPI-conflict-fop-store-local-yes.c	FN	FN	FN
013-MPI-conflict-fop-load-local-yes.c	FN	FN	FN
014-MPI-conflict-cas-store-local-yes.c	FN	FN	FN
015-MPI-conflict-cas-load-local-yes.c	FN	FN	FN
016-MPI-conflict-get-load-remote-no.c	TN	TN	-
017-MPI-conflict-get-get-remote-no.c	TN	TN	-
018-MPI-conflict-get-store-remote-yes.c	TP	FN	-
019-MPI-conflict-get-put-remote-yes.c	TP	FN	-
020-MPI-conflict-get-gaccread-remote-no.c	TN	TN	-
021-MPI-conflict-get-acc-remote-yes.c	FN	FN	-
022-MPI-conflict-put-load-remote-yes.c	TP	TP	-
023-MPI-conflict-put-store-remote-yes.c	TP	TP	-
024-MPI-conflict-put-put-remote-yes.c	FN	FN	-
025-MPI-conflict-put-gaccread-remote-yes.c	FN	FN	-
026-MPI-conflict-put-acc-remote-yes.c	FN	FN	-
027-MPI-conflict-acc-load-remote-yes.c	FN	TP	-
028-MPI-conflict-acc-store-remote-yes.c	FN	TP	-
029-MPI-conflict-acc-acc-remote-no.c	TN	TP	-
030-MPI-conflict-acc-gaccread-remote-no.c	TN	TN	-
031-MPI-conflict-gaccread-gaccread-remote-no.c	TN	TN	-
032-MPI-conflict-gaccread-load-remote-no.c	TN	TN	-
033-MPI-conflict-gaccread-store-remote-yes.c	FN	FN	-
034-MPI-conflict-gacc-store-remote-yes.c	FN	FN	-
035-MPI-conflict-gacc-gacc-remote-no.c	TN	TN	-
036-MPI-conflict-fop-fop-remote-no.c	TN	TN	-
037-MPI-conflict-fop-store-remote-yes.c	FN	FN	-
038-MPI-conflict-cas-store-remote-yes.c	FN	FN	-
039-MPI-conflict-cas-cas-remote-no.c	TN	TN	-

Table 7: Results for Synchronization, Atomic, Hybrid categories

Test Name	MUST-RMA	PARCOACH-dynamic	PARCOACH-static
001-MPI-sync-fence-local-yes.c	TP	TP	TP
002-MPI-sync-fence-local-no.c	TN	TN	TN
003-MPI-sync-lock-local-yes.c	TP	TO	TP
004-MPI-sync-lock-local-no.c	TN	TO	FP
005-MPI-sync-lock-flush-local-yes.c	TP	TO	TP
006-MPI-sync-lock-flush-local-no.c	TN	TO	TN
007-MPI-sync-lockall-flushlocalall-local-yes.c	TP	TP	TP
008-MPI-sync-lockall-flushlocalall-local-no.c	TN	FP	FP
009-MPI-sync-request-local-yes.c	TP	FN	FN
010-MPI-sync-request-local-no.c	TN	TN	TN
011-MPI-sync-pscw-local-yes.c	TP	FN	TP
012-MPI-sync-pscw-local-no.c	TN	TN	FP
013-MPI-sync-lockall-flushall-remote-no.c	TN	TN	-
014-MPI-sync-lockall-flushall-remote-yes.c	TP	FN	-
015-MPI-sync-lockall-barrier-remote-no.c	TN	FP	-
016-MPI-sync-lockall-barrier-remote-yes.c	TP	FN	-
017-MPI-sync-lockall-remote-yes.c	TP	FN	-
018-MPI-sync-fence-3procs-remote-yes.c	TP	TP	-
019-MPI-sync-fence-3procs-remote-no.c	TN	FP	-
020-MPI-sync-lock-barrier-nonconsistent-remote-yes.c	TP	TO	-
021-MPI-sync-lock-barrier-remote-yes.c	TP	TO	-
022-MPI-sync-lock-barrier-remote-no.c	TN	TO	-
023-MPI-sync-lock-barrier-sameorigin-remote-no.c	TN	TO	-
024-MPI-sync-lock-barrier-sameorigin-remote-yes.c	TP	TO	-
025-MPI-sync-lock-flushlocal-sameorigin-remote-yes.c	TP	TP	-
026-MPI-sync-lock-flushlocal-sameorigin-remote-no.c	TN	FP	-
027-MPI-sync-lock-exclusive-remote-no.c	TN	TO	-
028-MPI-sync-lock-exclusive-3procs-remote-no.c	TN	TO	-
029-MPI-sync-lock-exclusive-remote-yes.c	TP	TO	-
030-MPI-sync-lock-sendrecv-remote-yes.c	TP	TO	-
031-MPI-sync-lock-sendrecv-remote-no.c	TN	TO	-
032-MPI-sync-lock-sendrecv-3procs-remote-no.c	FP	TO	-
033-MPI-sync-lock-sendrecv-3procs-remote-yes.c	TP	TO	-
034-MPI-sync-pscw-remote-no.c	TN	TN	-
035-MPI-sync-pscw-remote-yes.c	TP	FN	-
036-MPI-sync-polling-remote-yes.c	TP	FN	-
001-MPI-atomic-customdatatype-remote-no.c	TN	FP	-
002-MPI-atomic-customdatatype-remote-yes.c	FN	TP	-
003-MPI-atomic-disp-remote-yes.c	FN	FN	-
004-MPI-atomic-disp-remote-no.c	TN	FP	-
005-MPI-atomic-short-int-remote-yes.c	FN	FN	-
006-MPI-atomic-float-int-remote-yes.c	FN	FN	-
007-MPI-atomic-float-int-sameorigin-remote-yes.c	FN	FN	-
008-MPI-atomic-double-float-remote-yes.c	FN	FN	-
009-MPI-atomic-int-int-remote-no.c	TN	FP	-
010-MPI-atomic-int-int-sameorigin-remote-no.c	TN	FP	-
001-MPI-hybrid-master-local-yes.c	TP	TP	TP
002-MPI-hybrid-master-local-no.c	TN	TO	FP
003-MPI-hybrid-single-local-yes.c	TP	TP	TP
004-MPI-hybrid-single-local-no.c	TN	TO	FP
005-MPI-hybrid-ordered-local-no.c	TN	TO	FP
006-MPI-hybrid-for-local-yes.c	TP	TP	TP
007-MPI-hybrid-section-local-yes.c	TP	TP	TP
008-MPI-hybrid-section-local-no.c	TN	TO	FP
009-MPI-hybrid-task-local-yes.c	TP	TO	FN
010-MPI-hybrid-task-local-no.c	TN	TO	FP
011-MPI-hybrid-master-remote-yes.c	TP	FN	-
012-MPI-hybrid-master-remote-no.c	TN	FP	-
013-MPI-hybrid-single-remote-yes.c	TP	FN	-
014-MPI-hybrid-single-remote-no.c	TN	FP	-
015-MPI-hybrid-task-remote-yes.c	TP	FN	-
016-MPI-hybrid-task-remote-no.c	TN	FP	-
017-MPI-hybrid-section-remote-yes.c	TP	FN	-
018-MPI-hybrid-section-remote-no.c	TN	FP	-
019-MPI-hybrid-ordered-remote-no.c	TN	FP	-
020-MPI-hybrid-for-remote-yes.c	TP	FN	-
021-MPI-hybrid-section-barrier-origin-remote-yes.c	FN	TO	-
022-MPI-hybrid-section-sendrecv-origin-remote-yes.c	FN	TO	-