On-the-Fly Data Race Detection for MPI RMA Programs with MUST

Simon Schwitanski*, Joachim Jenke*, Felix Tomski*, Christian Terboven* and Matthias S. Müller*
*Chair for High Performance Computing, IT Center
RWTH Aachen University, Aachen, Germany
{schwitanski, jenke, tomski, terboven, mueller}@itc.rwth-aachen.de

***Abstract*—MPI Remote Memory Access (RMA) provides a one-sided communication model for MPI applications. Ensuring consistency between RMA operations with synchronization calls is a key requirement when writing correct RMA codes. Wrong API usage may lead to concurrent modifications of the same memory location without proper synchronization resulting in data races across processes. Due to their non-deterministic nature, such data races are hard to detect. This paper presents MUST-RMA, an on-the-fly data race detector for MPI RMA applications. MUST-RMA uses a race detection model based on happened-before and consistency analysis. It combines the MPI correctness tool MUST with the race detector ThreadSanitizer to detect races across processes in RMA applications. A classification quality study on MUST-RMA with different test cases shows a precision and recall of 0.95. An overhead study on a stencil and a matrix transpose kernel shows runtime slowdowns of 3x to 20x for up to 192 processes.**

***Index Terms*—MPI RMA, correctness, data race, MUST**

I. Introduction

The Message Passing Interface (MPI) [1] is the de-facto standard for distributed-memory programming in HPC. Its traditional message-passing communication model is two-sided: To exchange data, *both* the sending and the receiving process have to actively call MPI functions (send/recv). In MPI Remote Memory Access (RMA) [1, §12], also called *one-sided communication*, only one of the processes is actively involved in the communication: A process can access the memory of other processes remotely using read, write, and update calls. MPI RMA communication calls map directly to the Remote Direct Memory Access (RDMA) functionality provided by network interconnects as InfiniBand and can therefore offer a performance advantage over two-sided MPI communication [2].

In MPI RMA, the developer has to ensure consistency of remote memory operations explicitly with specific synchronization calls. The wrong usage of such calls might result in concurrent modifications of the same memory location leading to undefined behavior of the program as shown in Figure 1. We call that situation a *data race across processes*. Since such data races behave non-deterministically, they are often difficult to detect without tool support.

On-the-fly data race detectors like ThreadSanitizer [3] are commonly used tools for shared-memory programs to find races at runtime. Translating the detection concepts of those on-the-fly race detectors to data races across processes in RMA programs is, however, difficult since (1) the distributed-memory environment makes the exchange of analysis data more challenging, (2) capturing the causality of events in MPI processes is complex, and (3) the consistency semantics of MPI RMA is more complex than of shared-memory models. A potential race detection tool has to consider all these points to be correct and scalable.

In this paper, we present *MUST-RMA*, an on-the-fly race detector for MPI RMA programs, which uses the analysis infrastructure of the MUST correctness checking tool [4] in combination with the shared-memory race detector ThreadSanitizer. MUST-RMA intercepts all MPI calls of an application to detect the so-called *concurrent regions of remote memory accesses* that represent the execution timeframe in which a memory access of an MPI RMA operation can take place. The information on the concurrent regions is then annotated in the shared-memory race detector ThreadSanitizer which performs the actual race detection.

We make the following contributions:

- A classification of data races for MPI RMA programs that distinguishes *local buffer races* and *remote races*,
- a race detection model for MPI RMA programs that detects the concurrent regions of RMA operations based on happened-before and consistency analysis, and
- a prototype implementation of our tool MUST-RMA that implements the race detection model by combining the analysis capabilities of MUST and ThreadSanitizer.

The paper is structured as follows: In Section II, we give an introduction to the MPI RMA programming model. In Section III, we present a classification of data races in MPI RMA. Section IV presents the race detection model. In Section V, we

P0 (origin)	P1 (target)
	`win location X`
`Barrier`	`Barrier`
`Win_lock(P1)`	
`...`	
`buf = 42`	`...`
`Put(&buf,P1,X)`	**`X = 1`**
`...`	`...`
`Win_unlock(P1)`	
`Barrier`	`Barrier`

Fig. 1. Data race across processes in MPI RMA: P0 performs a remote write to location `X` using an `MPI_Put` call that is concurrent to the `X = 1` instruction at P1. The value of the variable `X` is undefined due to a data race.

 (DOI: 10.1109/Correctness56720.2022.00009)

describe the functionality of our analysis tool MUST-RMA and evaluate its classification quality and overhead in Section VI. Finally, we discuss related works on RMA race detection in Section VII and conclude the paper in Section VIII.

II. MPI Remote Memory Access

This section recaps the most important concepts of MPI Remote Memory Access (RMA) [1, §12]. Following the MPI terminology, we denote the process issuing the actual communication operation as *origin*, while we denote the process whose memory is accessed as *target*.

A. Windows

Before other processes can access any memory remotely, a process has to expose the desired memory region to be accessed from remote in a *window*. The window creation itself is done in a collective call (e.g., `MPI_Win_create`), whereby each process specifies the memory region it wants to expose. MPI returns an opaque window object that can be used in subsequent RMA communication and synchronization calls. When all RMA communication is completed, a call to `MPI_Win_free` frees the window.

B. Communication Calls

MPI RMA specifies three different variants of communication calls: *Put* calls read data from a given buffer at the origin and perform a remote write at the target. *Get* calls read data from the target remotely and write the result into a given buffer. *Accumulate* calls update a value at the target using a given update operation and guarantee atomicity at the granularity of the predefined datatype [1, §12.7.1]. All RMA communication calls are *non-blocking*, an RMA operation has to be completed with a synchronization call, as discussed in the next section.

C. Synchronization Calls and Completion Semantics

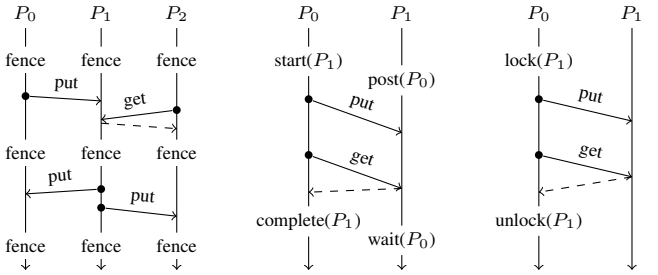
Before the origin can issue any RMA communication call, it first has to open an *access epoch* to an RMA window with an RMA *synchronization call*. Then, multiple RMA communication calls may follow. Finally, the epoch is completed with another synchronization call which enforces the completion of all RMA operations previously issued in that epoch. RMA distinguishes *origin completion* meaning that the local communication buffer at the origin can be reused and *target completion* meaning that the effect of the RMA operation is guaranteed to be visible at the target. RMA provides three kinds of synchronization mechanisms (fences, post-start-complete-wait, locks) as visualized in Figure 2 that can be categorized into two classes of synchronization, active target and passive target.

In *active target synchronization*, both origin and target are involved in ensuring completion. Corresponding to the access epoch at the origin, the target starts and ends an *exposure* epoch with synchronization calls. While the access epoch represents the period in which the origin issues RMA operations, the exposure epoch is the period *at the target* in which the RMA operations issued by the origin take place. Closing an access epoch implies origin completion while closing a matching exposure epoch implies target completion. MPI RMA defines two active target synchronization methods:

1) Fences: Processes synchronize *collectively* using the call `MPI_Win_fence` on the window. The call opens an access epoch at the origin and a matching exposure epoch at the target. Another call to that function implies the origin and target completion of all RMA operations issued in that epoch. Simultaneously, it opens the subsequent access and exposure epoch.

2) Post-Start-Complete-Wait (PSCW): A generalized variant of active target synchronization: The calls `MPI_Win_start` and `MPI_Win_complete` open and close an access epoch at the origin. The calls `MPI_Win_post` and `MPI_Win_wait` open and close an exposure epoch at the target. PSCW allows for fine-grained completion control compared to the bulk synchronization of fences.

In *passive target synchronization*, only the origin is involved in the RMA completion, and no exposure epochs are defined. The target does not have to invoke any synchronization call. The origin uses `MPI_Win_lock` and `MPI_Win_unlock` to open and close an access epoch, respectively. Closing an access epoch with `MPI_Win_unlock` implies both origin and target completion. Locks can be exclusive or shared and provide synchronized access to the RMA windows. MPI additionally offers `MPI_Win_flush` to ensure operation completion without unlocking the window. Further, `MPI_Win_lock_all` and `MPI_Win_unlock_all` enable access to the RMA window on *all* processes using a shared lock. Multiple processes may call it concurrently and can then issue RMA operations to all other processes. This *lock-all* model is closest to other PGAS models as OpenSHMEM [5].



(a) Fences (active) (b) PSCW (active) (c) Locks (passive)

Fig. 2. RMA synchronization modes as defined in [1, §12].

D. Memory Models

MPI RMA provides two kinds of memory models, the *separate* and the *unified* model. The separate memory model introduced in MPI-2 assumes a system where coherence between a private and a public window copy has to be established manually through additional RMA calls. In contrast, the unified model introduced in MPI-3 assumes a hardware-based coherence model. In this paper, we focus on the unified memory model.

Simon Schwitanski, Joachim Jenke, Felix Tomski, Christian Terboven, Matthias S. Müller:
On-the-Fly Data Race Detection for MPI RMA Programs with MUST.
2022 IEEE/ACM Sixth International Workshop on Software Correctness for HPC Applications (Correctness), IEEE, pp. 27-36
DOI: 10.1109/Correctness56720.2022.00009

III. Races in MPI RMA

In MPI RMA, origin and target completion must be ensured by using RMA synchronization calls to avoid data races with local memory accesses or other RMA operations. We distinguish two kinds of races: The *local buffer race* refers to a local conflict at the origin with other local memory accesses. The *remote race* is a conflict situation across processes, i.e., a remote access at the target conflicts with either (1) a local access at the target or (2) another remote access. This section discusses both race classes and follows the RMA semantics defined in [1, §12.7].

A. Local Buffer Races

For each RMA communication call, the user has to specify at the origin a local buffer from which the data to be transmitted is read in the case of *put* and *accumulate* operations or in which the resulting data is written in the case of *get* operations. After issuing an RMA communication call, a process should not do any write access to the local buffer in the case of *put* or *accumulate* operations and should not access the local buffer at all in the case of *get* operations until the corresponding operation is origin-completed. Otherwise, the behavior of the application is undefined [1, §12.7]. Figure 3 shows two examples of such buffer races. Conflicts can occur either between the buffer access and (1) a local memory access or (2) a buffer access from another RMA call that uses the same local buffer. Such a local buffer race occurs at the origin process that issues the RMA communication call. Local buffer races are not limited to RMA but are also possible in other non-blocking MPI calls that require a local buffer, e.g., non-blocking communication or non-blocking collectives.

B. Remote Races

Conflicts between a remote access *at the target* with other local or remote memory accesses are the second class of races. The RMA compatibility matrix depicted in Table I shows which kinds of operations at the target are valid to be performed concurrently to the *same* memory location and which are in conflict. Performing at least two of such conflicting memory operations concurrently without proper synchronization will lead to a *remote race*: *Get* operations trigger a remote read at the target; thus, a concurrent *load* access at the target to the same location is safe, while a concurrent *store* access will lead to a race. Similarly, *put* and *accumulate* operations trigger a remote write or update and thus *any* local memory access at the target will be in conflict. Multiple *get* operations to the same target location are safe, while any other combination of RMA operations to the same target location leads to a conflict. The only exception is the use of *accumulate* operations that follows the atomicity semantics as defined in the MPI standard [1, §12.7.1].

TABLE I
Compatibility of RMA operations at the target to the same memory location, adapted from [6]

	Load	**Store**	**Get**	**Put**	**Acc**
Load	–	–	safe	conflict	conflict
Store	–	–	conflict	conflict	conflict
Get	safe	conflict	safe	conflict	conflict
Put	conflict	conflict	conflict	conflict	conflict
Acc	conflict	conflict	conflict	conflict	safe*

*when adhering to the atomicity semantics of MPI RMA

Figure 4 shows different examples of remote races. While in Figure 4a and Figure 4b, a remote access conflicts with a local memory operation at the target, in Figure 4c, two remote accesses conflict at the target memory location without the target being involved at all. Due to the complexity of the RMA model, there are many other kinds of access patterns leading to race situations that are not depicted here.

C. Benign Races

Not all remote races in MPI RMA lead to undefined behavior [1, §12.7]. For example, reading locally from a memory location (polling of a state change of a memory location) that is concurrently accessed by a remote write or update is valid. Still, there is no guarantee on the ordering or atomicity of the accesses. Depending on the semantics of the program, a race might therefore be *benign*. But since this is a question of program semantics, we do not distinguish between benign and malicious races in our race detection.

P0 (origin)	P1 (target)
`Barrier`	`Barrier`
`Win_lock(P1)`	
`buf = 42`	...
`Put(&buf,P1,_)`	
`buf += 1`	
`Win_unlock(P1)`	
`Barrier`	`Barrier`

(a) Between put and local update

P0 (origin)	P1 (target)
`Win_fence`	`Win_fence`
...	
`buf = 42`	
`Put(&buf,P1,_)`	...
`Get(&buf,P1,_)`	
...	
`Win_fence`	`Win_fence`

(b) Between put and get

Fig. 3. Local buffer race examples. Conflicting statements are bold.

P0 (origin)	P1 (target)
	`win location X`
`Barrier`	`Barrier`
`Win_lock(P1)`	
...	
`buf = 42`	...
`Put(&buf,P1,X)`	**`X = 1`**
...	...
`Win_unlock(P1)`	
`Barrier`	`Barrier`

(a) Between put and local store

P0 (origin)	P1 (target)
	`win location X`
`Win_start(P1)`	
	`Win_post(P0)`
...	
`Get(&buf,P1,X)`	...
...	**`X = 1`**
	...
`Win_complete(P1)`	
	`Win_wait(P0)`

(b) Between get and local store

P0 (origin)	P1 (target)	P2 (origin)
	`win location X`	
`Barrier`	`Barrier`	`Barrier`
`Win_lock(P1,shared)`		`Win_lock(P1,shared)`
...		
`buf = 42`		`buf = 1`
`MPI_Put(&buf,P1,X)`	...	**`MPI_Put(&buf,P1,X)`**
...		
`Win_unlock(P1)`		`Win_unlock(P1)`
`Barrier`	`Barrier`	`Barrier`

(c) Between two put calls from different origin processes

Fig. 4. Remote race examples. Conflicting statements are bold.

Simon Schwitanski, Joachim Jenke, Felix Tomski, Christian Terboven, Matthias S. Müller:
On-the-Fly Data Race Detection for MPI RMA Programs with MUST.
2022 IEEE/ACM Sixth International Workshop on Software Correctness for HPC Applications (Correctness), IEEE, pp. 27-36
DOI: 10.1109/Correctness56720.2022.00009

IV. Race Detection Model

Reasoning on data races in MPI RMA requires capturing (1) the completion semantics of RMA operations with the consistency relation $\xrightarrow{co}$ and (2) the process synchronization of MPI processes with the happened-before relation $\xrightarrow{hb}$. In this section, we describe how to use those two relations to determine the so-called *concurrent regions* of RMA operations that represent the execution timeframe in which the access of an RMA operation can take place. The notion of a concurrent region of RMA operations was introduced by [7] but has not been formalized so far using the $\xrightarrow{co}$ and $\xrightarrow{hb}$ relation to reason on data races. For simplicity, we assume in this section that all RMA operations use the same window object.

A. Formal Semantics of MPI RMA

The formal semantics of MPI RMA has been defined in [2] in detail. Assuming N processes $P_0, \dots P_{N-1}$, we interpret the program execution as set of ordered events $E_i := \{e_0^i, e_1^i, \dots\}$ for each process P_i. The set of all events is $E := E_0 \cup \dots \cup E_{N-1}$. Events might be local memory accesses, MPI RMA calls, or any MPI synchronization call.

To detect data races between the different memory accesses and RMA operations, capturing the consistency relation $\xrightarrow{co}$ and happened-before relation $\xrightarrow{hb}$ is required. The *consistency relation* $\xrightarrow{co}$ captures the visibility of memory accesses: If $a \xrightarrow{co} b$, then the effect of a is guaranteed to be visible to b. In particular, if event a is an MPI RMA operation, then $a \xrightarrow{co} b$ means that the memory access of a is guaranteed to be finished before b in terms of MPI RMA completion semantics. The *happened-before relation* $\xrightarrow{hb}$ is a partial order capturing the causality of events in a program execution and is defined as follows:

Definition 1 (adapted from [8]). *The happened-before order $\xrightarrow{hb} \subseteq E \times E$ is the smallest transitive relation satisfying the following two conditions:*

1) *If $a, b \in E$ occur in the same process P_i and a is before b in program order, then $a \xrightarrow{hb} b$.*
2) *If a is a signal event and b is the corresponding wait event, then $a \xrightarrow{hb} b$.*

The second condition captures all kinds of synchronization between processes, in our case, using MPI send/recv calls or collective synchronization with barriers. If both $a \not\xrightarrow{hb} b$ and $b \not\xrightarrow{hb} a$, then we call a and b *concurrent*, because no execution order is guaranteed.

Based on the $\xrightarrow{co}$ and $\xrightarrow{hb}$ relation, we derive the *consistent happened-before relation* $\xrightarrow{cohb}$ as defined by [2]:

$$a \xrightarrow{cohb} b := a \xrightarrow{co} b \wedge a \xrightarrow{hb} b$$

Formally, a *data race* is present in an RMA program execution if for two *conflicting* (remote) memory access events a and b, it is $a \not\xrightarrow{cohb} b$ and $b \not\xrightarrow{cohb} a$. For simplicity, we also write $a \mid\mid_{cohb} b$.

B. Vector Clocks

To capture the $\xrightarrow{hb}$ relation between the different events, we rely on vector clocks [8]. Each process P_i manages a *vector clock* $V_i \in \mathbb{N}^N$ where each value $V_i[j]$ represents the logical timestamp that process P_i stores for process P_j.

Definition 2 (adapted from [8]). *Let $P_0, \dots, P_{N-1}$ denote the processes of a distributed computation. The vector clock V_i of a 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 signal event s, P_i updates V_i as in (2) and sends its vector clock to the waiting process P_j.
(4) On a wait event w, P_i updates V_i as in (2), waits for a matching vector $V(w)$ and updates its current V_i as follows: $V_i := \max\{V_i, V(w)\}$.

We did an in-depth classification of all synchronization concepts present in MPI and how they map to the vector clock exchange in [9]. For an event $e \in E_i$, we denote with $V(e)$ the vector clock V_i that process P_i had immediately after it executed event e.

C. Identifying Concurrency of RMA Operations

Each RMA operation issues (1) the local buffer access, which is locally concurrent with other events in a certain execution timeframe at the origin, and (2) the remote access, which is concurrent with other events in a certain execution timeframe at the target. We identify the execution timeframes of those accesses, i.e., the earliest and latest possible point in time when it might happen, by introducing the concept of a concurrent region: A *concurrent region* of an RMA operation $r \in E$ is a tuple (s, t) of events $s, t \in E_i$ at a given process P_i. The events s and t denote the start and end of the concurrent access. For all events e taking place between s and t on P_i, there is no $\xrightarrow{cohb}$ ordering with RMA operation r:

$$\forall e \in E_i : s \xrightarrow{hb} e \xrightarrow{hb} t \implies r \mid\mid_{cohb} e$$

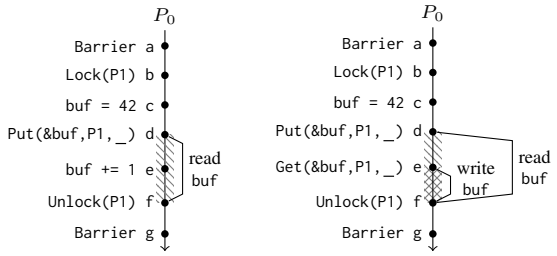
Each RMA operation has a concurrent region at the origin for the local buffer access and another concurrent region at the target for the remote access. In the following, we describe how to derive those concurrent regions.

D. Concurrent Region of Local Buffer Accesses

To detect local buffer races, the concurrent regions of local buffer accesses have to be identified. For that, the issued MPI RMA calls are analyzed: The buffer access can take place *at earliest* with the RMA call itself and *at latest* with the matching RMA synchronization call that guarantees origin completion. Other memory accesses in conflict with that local buffer access taking place *during* the concurrent region will therefore lead to a data race.

Figure 5a shows an example with a *put* call issued at the origin using the local buffer *buf*. Due to RMA completion semantics, the concurrent region of the *put* operation in event

Simon Schwitanski, Joachim Jenke, Felix Tomski, Christian Terboven, Matthias S. Müller:
On-the-Fly Data Race Detection for MPI RMA Programs with MUST.
2022 IEEE/ACM Sixth International Workshop on Software Correctness for HPC Applications (Correctness), IEEE, pp. 27-36
DOI: 10.1109/Correctness56720.2022.00009



(a) Concurrent region of put (b) Concurrent region of put and get

Fig. 5. Concurrent regions of local buffer accesses.

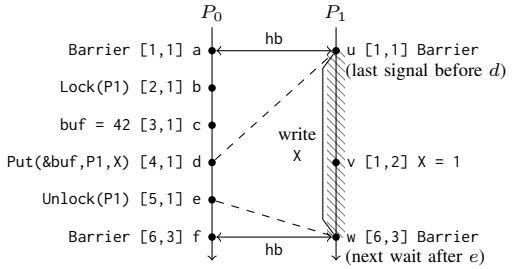


Fig. 6. Concurrent region of remote access with passive target communication and barrier synchronization.

d is (d, f). It starts with the *put* call itself and ends with the *unlock* call. The local buffer *buf* is modified during the concurrent region in event e. Since the local buffer access in event d is not guaranteed to be finished before event e and vice-versa, it holds $d \overset{co}{\not\to} e$ and $e \overset{co}{\not\to} d$, so $d \mid\mid_{cohb} e$. Thus, there is a local buffer race at process P_0. Similarly, Figure 5b shows a *put* call with a *get* call conflicting at the origin. Both use the same local buffer and their concurrent regions (d, f) and (e, f) are overlapping, so $d \mid\mid_{cohb} e$ implies a local buffer race at process P_0.

E. Concurrent Region of Remote Accesses

Detecting an RMA call's concurrent region at the target requires understanding both consistency and process synchronization. Intuitively, the remote access of an RMA operation issued at the origin P_o to target P_t can occur from P_t's point of view at earliest after the *latest preceding* synchronization event where P_t signaled to P_o. We name that event *last signal* and derive it based on the vector clocks:

Definition 3 (Last Signal)**.** *Let $a \in E_o$ be an event at P_o. The last signal of P_t to P_o before a, denoted as $LS_{t\to o}(a)$, is the event $b \in E_t$ where $V(a)[t] = V(b)[t]$.*

After target completion has been ensured at the origin, we have to find the latest point in time when target completion will be guaranteed in the view of the target. We define the *next wait* as the *earliest subsequent* synchronization event where P_t has to wait for P_o:

Definition 4 (Next Wait)**.** *Let $a \in E_o$ be an event at P_o. The next wait of P_t for P_o after a, denoted as $NW_{t\to o}(a)$, is the event $b \in E_t$ where*

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

Let $a \in E_o$ be any RMA call issued at the origin process P_o to target process P_t and let $c \in E_o$ be the corresponding RMA synchronization call issued at process P_o guaranteeing target completion. Then, the concurrent region of the remote access at P_t is $(LS_{t\to o}(a),\ NW_{t\to o}(c))$.

Figure 6 shows the vector clocks and the concurrent region for the race example from Figure 4a. Event d at P_0 starts the *put* operation. The earliest point in time when the remote write might occur from perspective of P_1 is immediately after the barrier in event u, because $LS_{1\to 0}(d) = u$. Target completion is guaranteed with the *unlock* call in event e and $NW_{1\to 0}(e) = w$, so the remote write will occur at latest from perspective of P_1 when it finishes the barrier in event w. The concurrent region of the *put* operation at target P_1 is (u, w). The conflicting local access `X=1` in event v is not ordered regarding the remote access, so $d \mid\mid_{cohb} v$ results in a race.

Figure 7 shows an example using active target communication with fences that is race-free. Processes P_0 and P_1 first synchronize with a fence which implicitly provides barrier synchronization. This will open a new access epoch at P_0 and an exposure epoch at P_1. Before P_0 issues the *put* operation in d, it waits for a message from P_1 in a *recv* call in b which is correspondingly reflected in the vector clock. For the *put* operation in event d, the last signal from P_1 is the *send* call, i.e., $LS_{1\to 0}(d) = w$. The next *fence* call in events e and x provides target completion and synchronization, therefore, it is $NW_{1\to 0}(e) = x$. The concurrent region is (w, x), therefore, the conflicting access `X=1` in event v is consistent happened-before-related, i.e., $v \xrightarrow{cohb} d$, so there is no race.

F. Limitations

The concurrent region is sometimes not enough to correctly detect races because two origin processes could synchronize externally the access to the same target, e.g., via exclusive locks on a window. From the target's point of view, however, the concurrent regions of those two memory accesses are overlapping and thus a race. Considering the example in Figure 4c, replacing the shared locks with exclusive locks would fix the

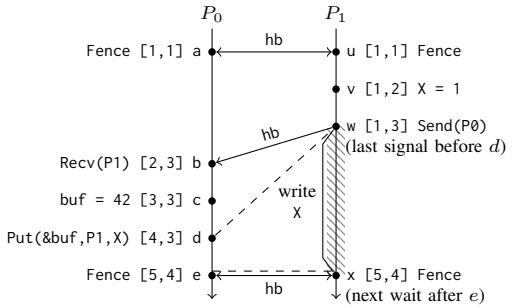


Fig. 7. Concurrent region of remote access using active target synchronization.

Simon Schwitanski, Joachim Jenke, Felix Tomski, Christian Terboven, Matthias S. Müller:
On-the-Fly Data Race Detection for MPI RMA Programs with MUST.
2022 IEEE/ACM Sixth International Workshop on Software Correctness for HPC Applications (Correctness), IEEE, pp. 27-36
DOI: 10.1109/Correctness56720.2022.00009

data race, but the concurrent regions of the two *put* calls would nevertheless be overlapping. In our implementation, we account for that where possible, e.g., by checking whether two conflicting RMA operations are protected via exclusive locks.

V. Implementing a Race Detector

We implemented the race detection model in the on-the-fly race detector MUST-RMA that we present in this section. It combines the MPI correctness tool MUST and the shared-memory race detector ThreadSanitizer to find RMA races.

A. MUST

MUST (Marmot Umpire Scalable Tool) [4] is an event-based runtime correctness checking utility for MPI applications. It detects MPI programming mistakes such as wrong parameters, resource leaks, deadlocks, type matching errors, and overlaps in communication buffers. MUST instruments MPI calls with P^{n}MPI [10] and uses the communication and tool infrastructure provided by the Generic Tools Infrastructure (GTI) [11]. GTI is a scalable framework for event-based parallel analysis tools. It can spawn additional tool processes or threads along with the actual MPI application to offload analysis computations. Analysis algorithms may run locally in the application or on a separate tool thread or tool process. The tool developer has to specify communication channels between the application process and the tool infrastructure. For that, different communication strategies (blocking or non-blocking exchange of analysis data) can be used.

For our RMA race analysis, we use one tool thread per application process in which we do all required analyses to determine the concurrent regions.

B. ThreadSanitizer

ThreadSanitizer (TSan) [3] is a dynamic data race detector for shared-memory programs using compile-time instrumentation relying on LLVM. TSan instruments memory accesses, memory allocation routines, and synchronization primitives to analyze for data races at program runtime. To store the current state of a memory location, each application memory location is mapped to a corresponding shadow memory cell. On every memory access, the state of a shadow memory cell is updated according to a state machine and checked for a race. The race detection itself relies on happened-before analysis based on vector clocks. TSan tracks synchronization primitives of libpthread and C++ threading natively. Through *dynamic annotations* [12], TSan can be made aware of any kind of thread synchronization by calling the functions `HappensBefore` and `HappensAfter` at runtime. A pair of those two calls at different threads establishes a happened-before relationship between those threads.

In recent versions, TSan supports *fibers* [13] to capture concurrency within a thread itself. Fibers can be dynamically annotated with API calls during execution. This makes it possible to create fibers as additional concurrent execution units in TSan's race detection model. In our RMA analysis, we use the fiber API to annotate concurrent regions.

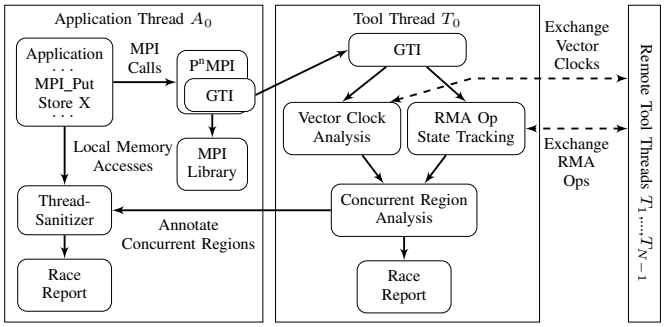


Fig. 8. Analysis workflow of MUST-RMA showing the analysis components for application thread A_0 and tool thread T_0.

We decided to rely on TSan instead of doing the actual race analysis directly in MUST since TSan is a mature and efficient tool for that. Further, since TSan already tracks thread synchronization, an extension to races in hybrid models as MPI+OpenMP in the future will be more straightforward.

C. Analysis Workflow

In the following, we assume that the application runs single-threaded with N processes. After compiling the application with TSan instrumentation, the actual analysis run can start: MUST spawns a tool thread within each application process that does all analyses and TSan annotations.

The on-the-fly analysis workflow for an RMA application is depicted in Figure 8 for application thread A_0 and tool thread T_0. During application execution, local memory accesses are tracked and processed by TSan. The MPI calls are intercepted by MUST via P^{n}MPI and passed via GTI to the tool thread T_0. The tool thread itself uses two analysis modules, (1) the vector clock module, which analyzes all MPI calls related to process synchronization ($\xrightarrow{hb}$) and (2) the RMA operation state tracking, which analyzes all MPI RMA communication and synchronization calls ($\xrightarrow{co}$). The analysis data of process synchronization and RMA operation consistency is exchanged with the responsible remote tool threads. The tool threads send the analysis data (vector clocks, RMA communication and synchronization calls) in a non-blocking fashion (where possible) via GTI to the destination tool threads. All tool threads periodically poll for incoming analysis information, so they will eventually receive the analysis data from other tool threads and process it.

Based on the information of process synchronization and RMA operation consistency, the concurrent region analysis module derives the concurrent regions according to the race detection model described in Section IV. When the tool thread detects the end of an RMA operation's concurrent region, its memory access is annotated to TSan as concurrent memory access in a fiber. TSan will check each annotated RMA memory access for races with (1) local memory accesses and (2) other annotated RMA memory accesses. In case of a race, it outputs a race report. The race report shows a stack trace with the source code locations of the two conflicting accesses.

Simon Schwitanski, Joachim Jenke, Felix Tomski, Christian Terboven, Matthias S. Müller:
On-the-Fly Data Race Detection for MPI RMA Programs with MUST.
2022 IEEE/ACM Sixth International Workshop on Software Correctness for HPC Applications (Correctness), IEEE, pp. 27-36
DOI: 10.1109/Correctness56720.2022.00009

D. Concurrent Regions as Fibers

The tool thread annotates concurrent regions as fibers using TSan's annotation API. During application execution, every MPI call that *could* be start of a concurrent region (RMA calls and any MPI synchronization call) is annotated by calling `HappensBefore(sc)`. This instructs TSan to store the current synchronization state with a given identifier `sc` and allows later to roll back the synchronization state to that point in time. When the tool thread detects the end of a concurrent region of an RMA operation, it (1) switches in TSan to a new fiber and (2) calls `HappensAfter(sc)` which sets the synchronization state of the fiber as if it ran at the start of the concurrent region with the identifier `sc`. Thus, TSan treats memory accesses annotated within the fiber as if they were done immediately at the start of the concurrent region. Then, the memory access of the RMA operation is annotated to the fiber. If there is an RMA race with other concurrent memory accesses, TSan will detect it. After the memory access annotation, the concurrent region is ended by switching back to the main thread. Thus, upcoming recorded memory accesses will not be in conflict with the accesses annotated in the concurrent region.

Fibers are dynamically created when they are needed to annotate concurrent regions. The RMA analysis reuses fibers to save resources: RMA operations from the same origin will be annotated using the same fiber. The downside is that remote races at a target due to multiple conflicting RMA operations from the *same* origin cannot be detected by TSan as they will be annotated to the same fiber. For that case, our RMA analysis checks for conflicts between remote accesses from the same origin directly in the concurrent region analysis.

VI. Evaluation

We evaluated both the classification quality and the overhead of MUST-RMA. The results are presented in this section.

A. Experiment Setup

We ran all measurements on the CLAIX18 [14] cluster, which consists of 1,200 nodes connected via Intel Omni-Path. Each node has 48 cores (two Intel Skylake Platinum 8160 processors) with SMT disabled and 192 GB of main memory. The benchmarks were compiled and instrumented with the Clang 12 compiler and run with Intel MPI 2018.

B. Classification Quality

To test the classification quality of MUST-RMA, we developed a microbenchmark suite consisting of small code examples with and without races. The test cases cover all three RMA synchronization modes (fences, PSCW, passive target) and different combinations of races between RMA calls and local memory accesses. In total, we analyzed 33 test cases with local buffer races and remote races. Roughly half of the test cases contains a race, while the other half does not. The MPI Bugs Initiative (MBI) [15] provides 48 RMA race test cases, where the *local concurrency* class covers local buffer races, and the *global concurrency* class covers remote races.

TABLE II
Classification Quality Benchmark Results

Test Case Class	Total	TP	FP	TN	FN	P	R
Own Tests - Local Buffer Race	10	5	0	5	0	1.00	1.00
Own Tests - Remote Race	23	14	1	7	1	0.93	0.93
MBI - Local Concurrency	36	18	0	18	0	1.00	1.00
MBI - Global Concurrency	12	12	0	0	0	1.00	1.00

Table II shows the results of running all test cases with MUST-RMA. We classify results as correct alerts (true positives, TP), false alerts (false positives, FP), error-free (true negatives, TN), and omission (false negatives, FN). Per class of test cases, we calculated the precision $P = \frac{TP}{TP+FP}$ and recall $R = \frac{TP}{TP+FN}$. The classification quality of MUST-RMA is promising: It could classify all RMA test cases from MBI correctly. In our own test suite, besides one FP and one FN, it classified all other cases correctly, leading to a total precision and recall of 0.95. For all correct alerts (TP), MUST-RMA reported the correct source code location of the data race.

1) False Positives: The false alert (FP) in our test suite is a limitation of the concurrent region approach as discussed in Section IV-F: The failing test case uses three processes, where two origin processes modify the same remote memory location of another target process. The two origin processes, however, synchronize themselves externally (without the target being involved) via send/recv calls, so there is no conflict at the target. Still, the target is not aware of that. Thus, MUST-RMA falsely detects a race in this case. To avoid the false alert, it would have to do further analyses on the vector clocks of the origin processes. Other false alerts might occur in happened-before analysis when a tool does not understand all kinds of process synchronization (collectives, send/recv pairs). MUST-RMA can track most MPI process synchronization through its vector clock analysis. For a discussion on supported synchronization patterns, we refer to [9].

2) False Negatives: As with every race detector using happened-before analysis, MUST-RMA might miss races if a non-deterministic interleaving of synchronization calls makes the race unobservable. Thus, for multiple executions of the same program, it might be that the race is observed and reported in some runs, while in others it is missed. We included such a test case with a race in our test suite that is missed (FN).

C. Overhead

We evaluated the overhead of MUST-RMA with the RMA variants of the *Stencil* and *Transpose* kernels from the Parallel Research Kernels suite [16]. The *Stencil* kernel implements a 5-point stencil operation on a 2D square grid and uses RMA *put* calls with fence synchronization for the halo exchange. The *Transpose* kernel transposes a square matrix where each process is assigned a block of columns of the original matrix. Each process transposes its assigned block of columns locally and puts the required blocks to the other processes using passive target synchronization with `MPI_Win_lock_all`, and

Simon Schwitanski, Joachim Jenke, Felix Tomski, Christian Terboven, Matthias S. Müller:
On-the-Fly Data Race Detection for MPI RMA Programs with MUST.
2022 IEEE/ACM Sixth International Workshop on Software Correctness for HPC Applications (Correctness), IEEE, pp. 27-36
DOI: 10.1109/Correctness56720.2022.00009

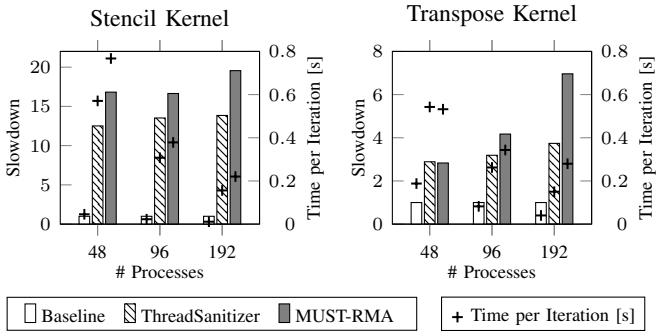


Fig. 9. Slowdown of average iteration time for the Stencil and Transpose kernel. 24 cores per node are used for the application. For the MUST-RMA run, we used the remaining 24 spare cores per node for the tool threads.

`MPI_Win_flush` calls. Both kernels are memory-bound and perform many RMA calls, so they are good candidates to evaluate the efficiency of MUST-RMA in worst-case scenarios.

The kernels were executed in three variants: (1) baseline without any instrumentation, (2) with TSan instrumentation only, and (3) with MUST-RMA (TSan + MUST analysis). We used 24 application processes per node with spread pinning. In the runs with MUST-RMA, we used the spare 24 cores of each node for the tool threads. The Stencil kernel was executed with 400 iterations and a matrix dimension of 20,000, while the Transpose kernel was executed with 400 iterations and a matrix dimension of 15,360.

Figure 9 shows the slowdown of the average iteration time compared to the baseline run for the ThreadSanitizer-only and MUST-RMA variant. The run with TSan has a slowdown of roughly 15x for the Stencil kernel and 4x for the Transpose kernel. Since the Stencil kernel has to read for each matrix element the neighboring elements, the higher TSan overhead due to the higher number of memory accesses is expected. Running with MUST-RMA increases the slowdown to 17x - 20x for the Stencil and 4x - 7x for the Transpose kernel. The slowdown expectedly increases with the number of processes, as (1) the total number of RMA calls of different processes to be exchanged by MUST-RMA increases and (2) the size of the vector clocks increases with the number of processes.

For a correctness checking tool, we believe that the overhead of MUST-RMA is reasonable. In future extensions, the TSan overhead can be significantly reduced by (1) performing static analysis to only instrument memory accesses relevant for the RMA race detection or (2) modifying TSan to ignore memory accesses at runtime that are not part of window memory.

VII. Related Work

Most of existing MPI correctness checking tools [4], [17]–[19] focus on MPI two-sided communication only. MPI RMA race detection has been the subject of research in a few publications: The approach in [20] detects races on-the-fly between RMA operations by managing the state of the RMA window locations in a mirror window as shadow memory. The approach can only detect races between RMA operations, not races with local memory accesses. MC-Checker [7] is a dynamic data race detector for MPI RMA programs. It traverses post-mortem the DAG of recorded memory accesses and synchronization events to detect the concurrent regions of RMA operations. MC-CChecker [21] improves the scalability of MC-Checker and reduces the number of false positives by using encoded vector clocks for the happened-before analysis. MC-Checker and MC-CChecker do post-mortem analyses, whereas MUST-RMA detects races at runtime. The approach in [22] performs an on-the-fly race analysis with binary search trees. Similar to our approach, it uses tool threads to transmit information about ongoing RMA calls and checks for overlapping memory accesses. The approach is limited to RMA programs which open and close RMA epochs collectively, i.e., all processes call `Win_lock/unlock_all` or `Win_fence` collectively. In our approach, we also consider fine-grained epoch synchronization via exclusive locks, `Win_flush` calls and synchronization through `MPI_Send/Recv` pairs.

Nasty-MPI [23] intercepts RMA communication calls on-the-fly and intentionally delays them to force synchronization errors that would not be observed in a usual execution. The tool itself, however, does not detect synchronization errors.

VIII. Conclusion

In this paper, we presented new concepts to detect data races in MPI RMA programs. Such data races can occur due to wrong usage of MPI RMA synchronization calls. We defined a race detection model that tracks MPI RMA operations' happened-before and consistency relation to determine their concurrent regions. Based on the race detection model, we have developed MUST-RMA, an on-the-fly race detector for MPI RMA programs combining the correctness checking tool MUST and the shared-memory race detector ThreadSanitizer. MUST-RMA shows a strong classification quality: It classifies all RMA race test cases of the MPI Bugs Initiative correctly and achieves in our own test suite a precision and recall of 0.95. The slowdown on a stencil and a matrix transpose kernel is 3x to 20x for up to 192 processes, where a significant portion is due to the memory tracing of ThreadSanitizer.

In future work, we will employ static analysis techniques to only instrument the memory accesses with ThreadSanitizer that could conflict with RMA operations to reduce the runtime overhead. This includes an evaluation of larger test cases with real-world HPC applications. Further, MUST-RMA provides only basic support for the detection of conflicts due to the wrong alignment in atomic operations through `MPI_Accumulate` calls, which we plan to extend soon. Although we tested only single-threaded MPI programs with MUST-RMA, we believe that an extension to hybrid programming models such as MPI+OpenMP is possible without much effort. Further, the race detection model itself is not limited to MPI RMA, so we are working on support of further one-sided communication models, e.g., PGAS models like OpenSHMEM, in future extensions.

The source code of MUST-RMA and evaluation results are available at https://doi.org/10.5281/zenodo.7129346.

Simon Schwitanski, Joachim Jenke, Felix Tomski, Christian Terboven, Matthias S. Müller:
On-the-Fly Data Race Detection for MPI RMA Programs with MUST.
2022 IEEE/ACM Sixth International Workshop on Software Correctness for HPC Applications (Correctness), IEEE, pp. 27-36
DOI: 10.1109/Correctness56720.2022.00009

REFERENCES

[1] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard Version 4.0." http://mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf, 2021. [online; accessed 12-October-2022].

[2] T. Hoefler, J. Dinan, R. Thakur, B. Barrett, P. Balaji, W. Gropp, and K. Underwood, "Remote Memory Access Programming in MPI-3," *ACM Transactions on Parallel Computing*, vol. 2, pp. 9:1–9:26, June 2015.

[3] K. Serebryany, A. Potapenko, T. Iskhodzhanov, and D. Vyukov, "Dynamic Race Detection with LLVM Compiler," in *Runtime Verification*, vol. 7186, pp. 110–114, Springer, 2012. Series Title: Lecture Notes in Computer Science.

[4] T. Hilbrich, M. Schulz, B. R. Supinski, and M. S. Müller, "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*, pp. 53–66, Springer, 2009.

[5] "OpenSHMEM: Application Programming Interface Version 1.5." http://openshmem.org/site/sites/default/site_files/OpenSHMEM-1.5.pdf, 2020. [online; accessed 12-October-2022].

[6] P. Balaji, W. Gropp, T. Hoefler, and R. Thakur, "Advanced MPI Programming." https://web.cels.anl.gov/~thakur/sc16-mpi-tutorial/slides.pdf. [online; accessed 12-October-2022].

[7] Z. Chen, J. Dinan, Z. Tang, P. Balaji, H. Zhong, J. Wei, T. Huang, and F. Qin, "MC-Checker: Detecting Memory Consistency Errors in MPI One-sided Applications," in *Proceedings of the 2014 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pp. 499–510, IEEE, 2014.

[8] R. Schwarz and F. Mattern, "Detecting causal relationships in distributed computations: In search of the holy grail," *Distributed Computing*, vol. 7, pp. 149–174, Mar. 1994.

[9] S. Schwitanski, F. Tomski, J. Protze, C. Terboven, and M. S. Müller, "An On-the-Fly Method to Exchange Vector Clocks in Distributed-Memory Programs," in *2022 International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 530–540, IEEE, 2022.

[10] M. Schulz and B. R. De Supinski, "P^nMPI tools: A whole lot greater than the sum of their parts," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pp. 30:1–30:10, ACM, 2007.

[11] T. Hilbrich, M. S. Müller, B. R. De Supinski, M. Schulz, and W. E. Nagel, "GTI: A Generic Tools Infrastructure for Event-Based Tools in Parallel Systems," in *Proceedings of the 26th International Parallel and Distributed Processing Symposium*, IPDPS '12, pp. 1364–1375, IEEE, 2012.

[12] K. Serebryany and T. Iskhodzhanov, "ThreadSanitizer: Data Race Detection in Practice," in *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pp. 62–71, ACM, 2009.

[13] "Fiber support for thread sanitizer." https://reviews.llvm.org/D54889. [online; accessed 12-October-2022].

[14] "CLAIX18: RWTH Compute Cluster." https://hpc.rwth-aachen.de/claix18. [online; accessed 12-October-2022].

[15] M. Laurent, E. Saillard, and M. 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)*, pp. 1–9, 11 2021.

[16] R. F. Van der Wijngaart and T. G. Mattson, "The Parallel Research Kernels," in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–6, IEEE, 2014.

[17] S. S. Vakkalanka, S. Sharma, G. Gopalakrishnan, and R. M. Kirby, "ISP: a tool for model checking MPI programs," *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 285–286, 2008.

[18] S. F. Siegel and T. K. Zirkel, "TASS: The Toolkit for Accurate Scientific Software," *Mathematics in Computer Science*, vol. 5, no. 4, pp. 395–426, 2011.

[19] "Intel Trace Analyzer and Collector." https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/trace-analyzer.html. [online; accessed 12-October-2022].

[20] M. Y. Park and S. H. Chung, "Detecting Race Conditions in One-Sided Communication of MPI Programs," in *Proceedings of the 2009 8th IEEE/ACIS International Conference on Computer and Information Science*, ICIS '09, pp. 867–872, IEEE, 2009.

[21] T.-D. Diep, K. Fürlinger, and N. 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, EuroMPI'18*, ACM, 2018.

[22] T. C. Aitkaci, M. Sergent, E. Saillard, D. Barthou, and G. Papauré, "Dynamic Data Race Detection for MPI-RMA Programs," in *EuroMPI '21 - European MPI Users' Group Meeting*, Sept. 2021.

[23] R. Kowalewski and K. Fürlinger, *Nasty-MPI: Debugging Synchronization Errors in MPI-3 One-Sided Applications*, pp. 51–62. Cham: Springer, Aug. 2016.

Simon Schwitanski, Joachim Jenke, Felix Tomski, Christian Terboven, Matthias S. Müller:
On-the-Fly Data Race Detection for MPI RMA Programs with MUST.
2022 IEEE/ACM Sixth International Workshop on Software Correctness for HPC Applications (Correctness), IEEE, pp. 27-36
DOI: 10.1109/Correctness56720.2022.00009

ARTIFACT DESCRIPTION

This artifact description explains how to reproduce the classification quality and the overhead evaluation results. In the following, we assume that `$ROOT` is the root folder of the supplemental repository / unpacked files. The required source code is available at https://doi.org/10.5281/zenodo.7129346.

A. Software Requirements

The following software packages are needed to reproduce the results:

- Clang compiler (preferably in version 12.0.1)
- MPI library with support for at least MPI 3.0 (preferably Intel MPI or MPICH)
- CMake in version 3.20 or newer
- libxml2 parser (libxml2-dev)
- Python 3

The classification quality benchmarks in addition need:

- LLVM lit in version 14.0.0 (available via PyPI)
- FileCheck binary (distributed with LLVM)

The overhead evaluation in addition needs:

- JUBE benchmarking environment in version 2.4.2 or newer (http://www.fz-juelich.de/jsc/jube)
- Slurm scheduler to submit the batch scripts

B. Classification Quality Benchmarks

To simplify the reproduction of the classification quality benchmarks, we provide a Dockerfile that provides the required software environment to build and run MUST-RMA with the benchmarks. If instead a cluster environment is used, the following Docker build and run steps can be skipped.

Build the docker image with tag `must-rma`, adjust permissions for the `must_rma` subfolder to match with the container user, and run the produced docker image with the MUST source code mounted as volume:

```
# cd $ROOT
# docker build docker -t must-rma
# chown -R 1000:1000 ./must_rma
# docker run --rm -it \
    -v $(pwd)/must_rma:/must_rma must-rma /bin/bash
```

Change to the `must_rma` directory. Install MUST-RMA by using the provided install script `build_must.sh`:

```
$ cd $ROOT/must_rma
$ ./build_must.sh
```

Build and installation path can be set within the script. In the following, we assume that MUST-RMA was built in the folder `$BUILD` and installed in `$INSTALL`.

Change into the `$BUILD` directory and run the tests:

```
$ cd $BUILD
$ lit -j 1 tests/OneSidedChecks/ | tee test_output.log
```

This runs all 81 test cases and outputs the results (number of passed and failed tests). Passed tests are marked as `PASS`, failed tests with `FAIL` or `XFAIL`. The number of workers (parameter `-j`) can be increased, however spawning too many workers might lead to failed test cases if there are not enough cores available to run the tests.

To produce the result table (Table II), we provide a Python script that parses the `test_output.log` file. Change back to the `classification_quality` folder and pass the test output log file to the script:

```
$ cd $ROOT/classification_quality
$ python3 generate_classification_quality_table.py \
    $BUILD/test_output.log
```

To run tests on own applications / binaries, MUST-RMA can be run with:

```
$ $INSTALL/bin/mustrun --must:distributed \
    --must:tsan --must:rma \
    -np <number of processes> <binary>
```

C. Overhead Evaluation

The overhead evaluation is specific to the CLAIX cluster, so running the benchmarks in another environment will need manual adaptations. We provide a JUBE configuration to make reproducibility easier. Important parameter sets within the JUBE configuration (`prk_rma.xml`) to consider:

- `prk_kernel_args_pset`: number of iterations and grid size to be used in the kernels
- `prk_system_pset`: system configuration, e.g., number of nodes to be used

After configuring all required parameters, the benchmarks can be run with

```
$ cd $ROOT/overhead_measurement
$ jube run prk_rma.xml -t kernel_name
```

where `kernel_name` can be `stencil` or `transpose`.

The JUBE configuration (1) builds MUST-RMA, (2) builds the chosen kernel with and without TSan instrumentation, (3) submits a Slurm job with the requested number of nodes that runs the three different configurations (plain, tsan, must-rma). After the Slurm jobs have finished, the results can be retrieved with

```
$ cd $ROOT/overhead_measurement
$ jube result -a bench_run --id <id of JUBE run>
```

This prints out the results (average iteration time per second per configuration) of Figure 9 as a table.

The results of the measurements presented in the paper are available at `$ROOT/overhead_results`.

Simon Schwitanski, Joachim Jenke, Felix Tomski, Christian Terboven, Matthias S. Müller:
On-the-Fly Data Race Detection for MPI RMA Programs with MUST.
2022 IEEE/ACM Sixth International Workshop on Software Correctness for HPC Applications (Correctness), IEEE, pp. 27-36
DOI: 10.1109/Correctness56720.2022.00009