Rethinking Data Race Detection in MPI-RMA Programs

Radjasouria Vinayagame, Van Man Nguyen, Marc Sergent, Samuel Thibault, Emmanuelle Saillard

Rethinking Data Race Detection in MPI-RMA Programs

Radjasouria Vinayagame
Eviden
Echirolles, France
radjasouria.vinayagame@eviden.com

Van Man Nguyen
Eviden
Echirolles, France
van-man.nguyen@eviden.com

Marc Sergent
Eviden
Echirolles, France
marc.sergent@eviden.com

Samuel Thibault
University of Bordeaux
Bordeaux, France
samuel.thibault@u-bordeaux.fr

Emmanuelle Saillard
Inria
Bordeaux, France
emmanuelle.saillard@inria.fr

ABSTRACT

Supercomputers are capable of increasingly more computations, and nodes forming them need to communicate even more efficiently with each other. The Message Passing Interface (MPI) proposes a communication model based on one-sided communications called the MPI Remote Memory Access (MPI-RMA). Thanks to these operations, applications can improve the overlap of communications with computations. However, one-sided communications are complex to write since they are subject to data races. This paper rethinks an existing on-the-fly data race detection algorithm for MPI-RMA programs by improving the storage of memory accesses in a Binary Search Tree using a new insertion algorithm based on fragmentation and merging algorithms. Thus, experimental results on real-life applications show that this new insertion algorithm improves the accuracy of the data race detection and can reduce the overhead of the analysis at runtime by a factor up to two.

CCS CONCEPTS

• **Computing methodologies** → **Parallel programming languages**; • **Software and its engineering** → **Software testing and debugging**.

KEYWORDS

HPC, MPI-RMA, Verification, Data Race, Dynamic Analysis

ACM Reference Format:
Radjasouria Vinayagame, Van Man Nguyen, Marc Sergent, Samuel Thibault, and Emmanuelle Saillard. 2023. Rethinking Data Race Detection in MPI-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, 9 pages. https://doi.org/10.1145/3624062.3624086

1 INTRODUCTION

To meet the exascale challenge, parallel programming models tend to abstract the machine details with task-based programming or by exposing the hardware at the software level (network with RMA capabilities) with the Partitioned Global Address Space (PGAS) model. This model is based on one-sided communications which decouple data movement from synchronization.
The Message Passing Interface (MPI) standard proposes a similar communication model via MPI Remote Memory Access (MPI-RMA). With MPI-RMA, each MPI process makes a part of its memory available to other MPI processes so the latter can remotely read and write on this "distributed shared memory" with fewer synchronizations than two-sided MPI communications. Consequently, applications migrating from two-sided MPI to MPI-RMA operations should get significant speedup. Nevertheless, most MPI programs still use two-sided communications because MPI-RMA programs are challenging to write and error prone. Indeed, as in shared memory models, developers must ensure memory consistency in MPI-RMA programs to avoid data races. Thus, some tools aim at helping developers write correct and productive MPI-RMA applications to promote communication/computation overlap and asynchrony. However, these tools imply a noteworthy overhead at runtime induced by the analysis of the program and thus have scalability issues. This paper enhances an existing on-the-fly data race detection tool for MPI-RMA programs called RMA-Analyzer [1]. We propose a new algorithm that enhances its scalability and its accuracy.

The paper is organized as follows: Section 2 provides some background elements and the key concepts this work relies on. Section 3 lists the existing tools that can detect data races in MPI-RMA programs. Section 4 proposes a new algorithm that enhances the memory accesses management in RMA-Analyzer in order to reduce the overhead of the analysis. In Section 5, we compare our contribution against MUST-RMA, a state-of-the-art solution for the detection of data races in MPI-RMA programs. We highlight the improvements we describe in the paper and show a performance analysis. In Section 6, we discuss several subtleties of the MPI-RMA library that may impact the correctness of data race detection tools. Finally, Section 7 concludes this work.

2 BACKGROUND

This section describes how MPI-RMA works and the errors that can occur due to the specifics of the one-sided communications.

2.1 MPI-RMA

In MPI-RMA, each process has a "distributed shared memory" that can be remotely accessed by other MPI processes. In MPI-RMA,

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

ACM ISBN 979-8-4007-0785-8/23/11…$15.00
https://doi.org/10.1145/3624062.3624086

these memory regions are called *windows* and remote memory accesses to these windows are possible during an *epoch*. Within an epoch, MPI-RMA proposes several communication operations which involve two processes: the *origin* process which issues the MPI-RMA communication, and the *target* process whose window is accessed via the communication. In this paper, we focus on the Passive Target synchronization mode where only the *origin* process is involved in the synchronization. We consider the two major one-sided communication operations: `MPI_Put` which allows to write a value owned by the *origin* process to the window of a *target* process, and `MPI_Get` which allows the *origin* process to locally retrieve a value from the window of a *target* process. An example of all possible accesses within an epoch is shown in Figure 1. In the figure, gray parts represent the window. $P_i \rightarrow P_j$ means a communication from P_i to P_j. $outWin$ and $inWin$ respectively mean out of the window and in the window.

When using MPI-RMA, four types of memory accesses should be considered [1], depending on if the operation is local to the process ($Local\_*$) or if it is a remote memory access (RMA_*), and on if the operation is a $WRITE$ ($*\_Write$) or a $READ$ ($*\_Read$) operation. For instance, an `MPI_Put` operation is an RMA_Write for the *target* process and an RMA_Read for the *origin* process. Inversely, an `MPI_Get` operation is an RMA_Read for the *target* process and an RMA_Write for the *origin* process. A `Store` operation is a $Local\_Write$ while a `Load` operation is a $Local\_Read$.

MPI-RMA has shown performance improvement on applications that have migrated from MPI two-sided to MPI-RMA. As an example, *Mizan-RMA* [11], a graph processing framework, got a speedup up to 280% when using MPI-RMA compared to a previous version that uses `MPI_Send` and `MPI_Recv` operations. Similarly, the *Graph500* [12] data intensive benchmark got a speedup of 200%. The authors claim that these speedups are achieved through a better overlapping of communications and computations. However, writing efficient and correct MPI-RMA programs can be challenging. Especially because of the following three properties that are crucial to ensure an efficient overlap of communications with computations:

(1) *Completion*: since MPI-RMA communications are asynchronous, we cannot know if a communication has completed until the end of the epoch,
(2) *Ordering*: MPI-RMA communications can happen in any order within an epoch,
(3) *Atomicity*: the atomicity of MPI-RMA communications is only guaranteed at the *MPI_Datatype* level.

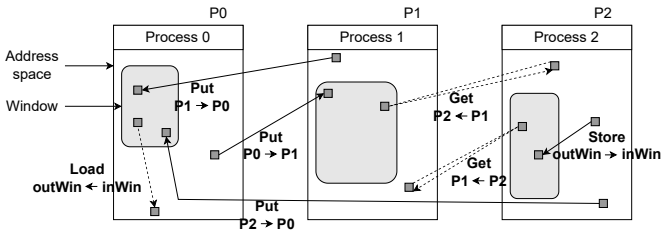


Figure 1: Possible memory access operations within an epoch.

2.2 Data Races in MPI-RMA Programs

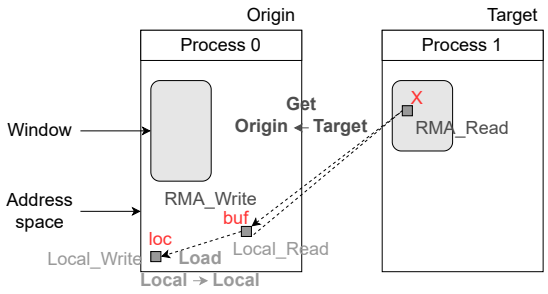
MPI-RMA programs expose memory through an abstraction that allows to read from and write to distant memory at any time, which can lead to data races. A data race occurs if two operations access the same memory range with at least one of them being an RMA access, and one of them being a $WRITE$ access (RMA_Write or $Local\_Write$). In such a scenario, the value in that memory space may change depending on the order in which the operations are executed. A more precise description of data races is presented by Hoefler et al. [6] and two examples of data races are presented in Figure 2. In Figure 2a, the *origin* process issues an `MPI_Get` operation to read the value of X at *target* side (RMA_Read) and update `buf` (RMA_Write). We suppose this operation is followed by a `Load` operation from `buf` to `loc`. Because of the completion property, a data race can occur at the *origin* side and `buf` is either equal to X or *loc*. Figure 2b shows a data race between two processes. In this example, both *origin* and *target* processes are reading and writing in their own window, and on overlapping address ranges.

Data races can also occur with more than two processes, for instance when multiple processes issue an `MPI_Put` operation to the same address space of the same target. All possible cases of data races are reported in Figure 3. Possible consistency errors within a process are represented by the first part of the table. We consider three processes: two that can initiate a communication (ORIGIN 1 and ORIGIN 2) and one target process (TARGET). Rows represent the first operation issued. The columns represent another operation issued either by the same process (ORIGIN 1) or another process (TARGET of the first operation or a new process issuing a communication, referred as ORIGIN 2). In each cell, the right bit refers to an error at origin side while the left bit refers to an error at target side. The example Figure 2a is represented by the cell (O1-GET, ORIGIN1-LOAD). "01" means that an error can occur only at origin side. Figure 2b is represented by the cell (O1-GET, TARGET-GET). Depending on if the value is read and written in or out of the window, an error can or cannot occur.

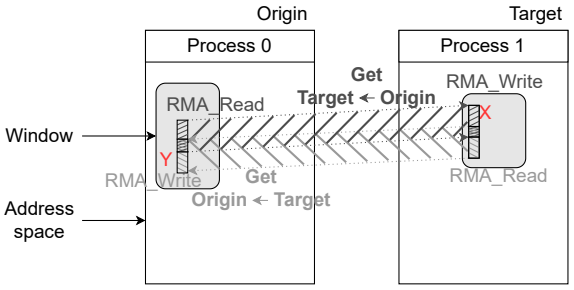
3 RELATED WORK

Many tools detect data races in multithreaded programs (e.g., Valgrind DRD [14], Helgrind [8] or ThreadSanitizer [18]). However, these tools cannot be directly applied to MPI-RMA programs because they are too general, so they struggle with the scale of MPI-RMA programs. On the other hand, tools that are aware of the semantics of MPI-RMA communications can significantly reduce the analysis workload.

Very few approaches exist to detect data races in MPI-RMA programs. Park et al. [15] presents an approach that creates a mirror window each time a window is created. Then, each time a new MPI-RMA communication accesses a memory space in the window, a check for data races is performed in the corresponding mirror window containing all previous accesses to that window. This approach does not consider local `Load` and `Store` accesses, thus leading to false negative results. Moreover, the implementation of this approach is only compatible with the MPI-2 standard and thus does not support the new MPI-RMA features such as the `MPI_Win_lock_all/MPI_Win_unlock_all` epoch creation functions. *MC-Checker* [2] uses a post-mortem analysis based on



(a) Data race at the origin process. The asynchronism of the `MPI_Get` operation makes the value of buf unpredictable.



(b) Data race occurring at origin and target sides. Dark grey hatching (resp. light grey) represents the remote read initiated by P1 (resp. P0) that stores the value of Y in X (resp. X in Y).

Figure 2: Examples of a data race within a process (left) and between processes (right).

		ORIGIN 1				TARGET				ORIGIN 2	
		GET	PUT	LOAD	STORE	GET	PUT	LOAD	STORE	GET	PUT
O1	GET	01	11	01	01	11/00	11/00	00	10	00	10
	PUT	11	10	00	01	11/00	11/00	10	10	10	10

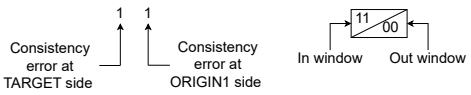


Figure 3: Data races situations with 3 processes. Rows represent the first operation, issued by a process origin 1 (O1). Columns represent another operation issued either by the same process (ORIGIN 1) or another process (TARGET or ORIGIN 2).

a Directed Acyclic Graph (DAG) to detect concurrent regions (in respect to the happens-before relation [10]). MC-Checker has been enhanced with *MC-CChecker* [3]. This new tool reduces the number of false positives and improves the scalability of MC-Checker through a clock-based approach based on the encoded vector clock. Like the work of Park et al., MC-CChecker is only compatible with the MPI-2 standard. *MUST-RMA* [17] is an on-the-fly data race detector for MPI-RMA programs which combines *MUST* [5] and *ThreadSanitizer*, a shared-memory data race detector [19]. MUST-RMA constructs concurrent regions based on the happens-before relation and forwards them to ThreadSanitizer which then checks for data races. A static approach detecting data races in MPI-RMA programs has been proposed by Saillard et al. [16]. The analysis performs a Breadth First Search (BFS) on the Control Flow Graph to detect data races at compile time. This method enables early detection of errors but is limited to errors occurring at the origin side only. To deal with performance, MC-Checker, MC-CChecker implementations have a lightweight static support that reduces the number of `Load`/ `Store` instrumentations. RMA-Analyzer [1] captures memory accesses during the execution of a program and stores them in a Binary Search Tree (BST). The implementation considers that a memory access includes information about the exact interval of addresses that are accessed (we are only considering consecutive accesses which means that all the addresses in the interval are accessed), the type of the access, and debug information (e.g., the location of the access in the source code). Thus, as soon as a data race is detected, RMA-Analyzer stops the program and returns an error message including debug information to facilitate the correction of the program. When an MPI window is created, each MPI process creates a BST. The BST is then filled with all memory locations the owner process or other processes accesses to (both remote accesses in the window and local accesses). When a new memory location is accessed, a first traversal of the BST is done in order to check for data races with the previous memory accesses. If there is no error, another traversal is done to insert the new access. We argue that this implementation of the approach may have performance bottlenecks, especially because of the size of the BST that is equal to the number of accesses in a program. Moreover, false negatives are possible because of the approximation made by only considering the lower bound of the interval of addresses when comparing two accesses. These limitations are due to the fact that accesses present in the BST are not disjoint, which leads to false negatives, and are not merged, which leads to slowdown. To solve these issues, we introduce a new insertion algorithm of memory accesses in the BST that increases the accuracy of the analysis, and reduces the memory footprint and overhead induced by the data race detection proposed by RMA-Analyzer.

4 NEW DATA RACE DETECTION

Our new data race algorithm improves the insertion of memory accesses in the BST. The algorithm is presented in Algorithm 1. Given a new memory access *newAcc*, the new insertion algorithm first checks for data races (line 2). If no data race is detected, the algorithm retrieves all the accesses present in *BST* that are intersecting with *newAcc* (line 5). Then, the fragmentation algorithm presented in Section 4.1 is called over all these accesses (line 6). Afterwards, the merging algorithm presented in Section 4.2 is called on $fragAcc$ to merge fragmented accesses if possible (line 7). Finally, `finish_insertion` is called to replace the "old" accesses by the "new" ones (line 8). An example of an insertion is presented in Figure 4. Step ① shows the state of the BST at the beginning of the algorithm. The colors correspond to different type of accesses. In step ②, only memory locations that intersect with *newAcc* are represented as the rest of the algorithm will only consider these accesses. The following subsections explain the fragmentation and

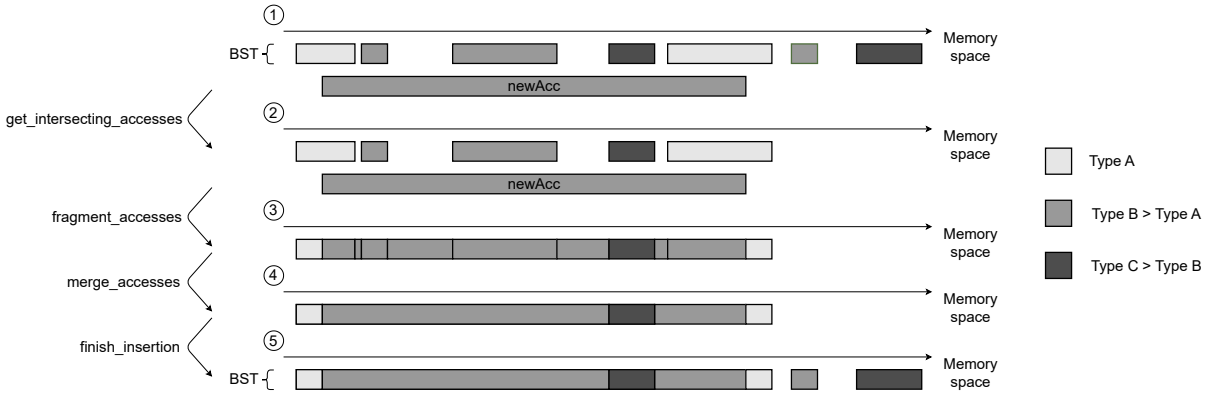


Figure 4: Illustration of the insertion of a new memory access in the BST.

Algorithm 1 Insertion of a memory access in the BST

```
1: function INSERT_BST(newAcc, BST)
2:     hasError ← data_race_detection(newAcc, BST)
3:                                  ▷ report an error in case of a data race
4:     if !hasError then
5:         interAcc ← get_intersecting_accesses(newAcc, BST)
6:         fragAcc ← fragment_accesses(interAcc, newAcc)
7:         mergedAcc ← merge_accesses(fragAcc)
8:         finish_insertion(interAcc, mergedAcc, BST)
```

merging algorithms in details (steps ③ and ④). Finally, step ⑤ updates the BST.

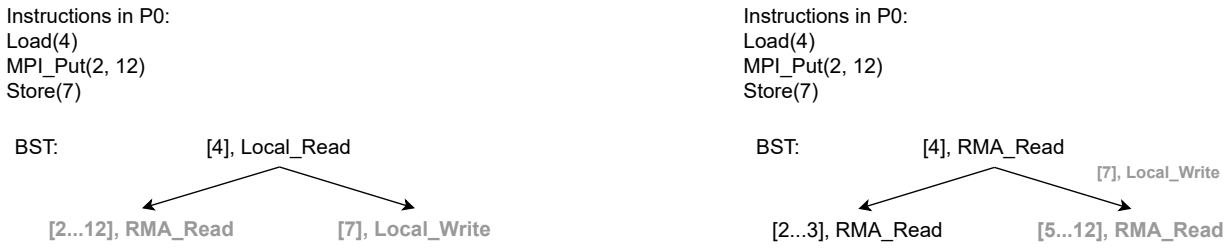
4.1 Improving the Accuracy of the Memory Accesses Insertion Algorithm

RMA-Analyzer can have false negatives caused by non-disjoint accesses. Indeed, when searching for accesses intersecting with a new access in the BST, RMA-Analyzer may miss some of these intersections and potentially miss a data race. This is due to the approximation made by only considering the lower bound of the interval of addresses to compare the accesses. An example of a code leading to a false negative is presented in Figure 8a. The corresponding BST created by RMA-Analyzer is presented in Figure 5a. In the figure, a node in the BST is noted (*memoryinterval*, *accesstype*) and instructions are given in the top left. In this example, the node corresponding to the `MPI_Put`, ([2...12], *RMA_Read*) is inserted in the left subtree of the node ([4], *Local_Read*). Then, when searching for the intersecting accesses with new access ([7], *Local_Write*), RMA-Analyzer does not notice the intersection between this new access and ([2...12], *RMA_Read*). Finally, the new access is inserted in the BST in the right subtree of ([4], *Local_Read*), without noticing the data race with ([2...12], *RMA_Read*). This false negative is a consequence of the fact that ([2...12], *RMA_Read*) and ([4], *Local_Read*) are not disjoint. To make the accesses disjoint in the BST, we propose a fragmentation algorithm that is called each time a new access is inserted in the BST. The algorithm works as follow: when a new access *new_acc* is inserted in the BST, if this *new_acc* is intersecting with another access *BST_acc* already present in the BST, three new accesses representing the parts of the intersection of the two intervals are created and inserted in the BST if they are not empty. The first access represents the leftmost subsection, that is not part of the intersection between *new_acc* and *BST_acc*. The second access represents the intersection of *new_acc* and *BST_acc*. Given the access type of *new_acc* and *BST_acc*, the access type and the debug information of the resulting access *intersection_frag* is given in Table 1. Basically, RMA accesses prevail on local accesses and *WRITE* accesses prevail on *READ* accesses. Therefore, if both accesses have the same access type, the debug information of the most recent access is kept. In the table, red cells represent cases where a data race may be detected if the second memory access is from another process. The third access *r_frag* represents the remaining subsection. An illustration of the fragmentation algorithm on two accesses is shown in Figure 6. In this figure, we consider that accesses with the same Type have the same access type and the same debug information. Using this new algorithm, the BST constructed by RMA-Analyzer presented in Figure 5a becomes the BST presented in Figure 5b.

This fragmentation algorithm has been implemented and tested on the program shown in Figure 8a. It successfully detected the data race between the bold statements, whereas RMA-Analyzer failed to raise the error. However, it can lead to a drastic increase of nodes in the BST. Indeed, each new access possibly increases the nodes in the BST by two : one node is removed and three nodes are added. This may lead to an explosion of the memory usage and slow down the analysis of the BST.

Table 1: Resulting access type and debug information given two accesses. Rows represent an access already inserted in the BST and columns represent the new access that is about to be inserted. *_R: *_Read, *_Write: *_Write, *-1/*-2: debug information of the $1^{st}/2^{nd}$ access, x: data race.

	Local_R-2	Local_W-2	RMA_R-2	RMA_W-2
Local_R-1	Local_R-2	Local_W-2	RMA_R-2	RMA_W-2
Local_W-1	Local_W-1	Local_W-2	RMA_R-2	RMA_W-2
RMA_R-1	RMA_R-1	x	RMA_R-2	x
RMA_W-1	x	x	x	x



(a) False negative with RMA-Analyzer. **(b) Error detected with our implementation.**

Figure 5: BST of code 1 with RMA-Analyzer (left) and using our fragmentation algorithm (right). A node contains the memory interval and the access type. Bold nodes are conflicting.

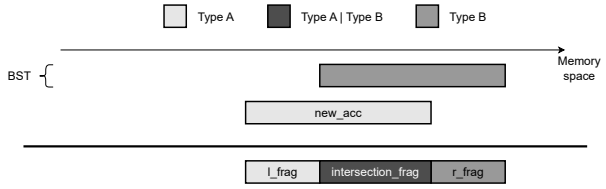


Figure 6: Illustration of an execution of the fragmentation algorithm given two Accesses.

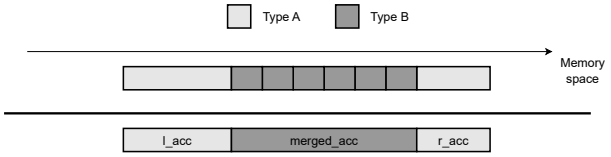


Figure 7: Illustration of an execution of the merging algorithm.

P0 (Origin)	P1 (Target)
Window location X	
Win_lock_all	Win_lock_all
temp = buf[4]	
Put(buf[2], 10, X)	
buf[7] = 1234	
Win_unlock_all	Win_unlock_all

(a) Code 1, reported as a false negative by RMA-Analyzer.

P0 (Origin)	P1 (Target)
Window location X	
Win_lock_all	Win_lock_all
for(i=0;i<1000;i++)	
Get(buf[i], 1, X)	
Get(buf[0], 1, X)	
Win_unlock_all	Win_unlock_all

(b) Code 2, with a one-sided communication in a loop.

Figure 8: Examples of codes to illustrate the fragmentation and merging algorithms.

4.2 Optimizing the Number of Nodes in the BST

In order to avoid the explosion of the number of nodes in the BST, which could negatively impact the scalability of the data race detection algorithm, we propose a node-merging algorithm.

To merge two nodes, two conditions need to be ensured: (1) the two accesses to be merged must be adjacent, and (2) they must have the same access type and debug information. Two accesses of the same access type cannot be merged if they have different debug information as they will not refer to the same instruction and will not be fixed in the same way. The merging algorithm goes through all the accesses created by the fragmentation algorithm and merges them if possible. Figure 7 illustrates an execution of the merging algorithm. In this figure, since the intervals of Type B are adjacent and are all of the same Type, meaning that they have the same access type and debug information, they can be merged. Figure 8b has multiple adjacent memory accesses : 5,002 memory accesses are made in this program. RMA-Analyzer creates a number of nodes in the BST that is linear in the number of iterations (1,000 in this case). Thus, the BST created by RMA-Analyzer has 5,002 nodes, each iteration adding five new nodes to the BST - variable *i* is read or written 4 times and *buf* is read once. The remaining two nodes are the first *LOCAL_READ* from `i=0` and the last *RMA_WRITE* from the `MPI_Get(buf[0],1,X)` instruction. The merging algorithm merges all the nodes induced by the `MPI_Get` communications into only one node in the BST. Indeed, these nodes are accessing adjacent memory spaces and have the same access type and debug information since they are called at the same line in the code. Thus, the merging algorithm updates the BST which is of size two: one node for the variable *i* and one node for all the `MPI_Get` accesses.

Regarding the complexity of the algorithm, since the most computationally intensive operations used in the new insertion algorithm are searches, insertions and deletions which are logarithmic in time as we use a (balanced) BST, our new insertion algorithm is also logarithmic in time.

5 EXPERIMENTAL RESULTS

This section first presents an overview of the RMA-Analyzer framework in which our approach has been implemented (Subsection 5.1). Then, Subsection 5.2 presents validation results on a microbenchmark suite we developed. In Subsection 5.3, we compare the overhead induced by our approach with MUST-RMA on two "real-life" applications: MiniVite and CFD-Proxy.

Our experiments were performed on an Eviden cluster that belongs to the Eviden R&D department, located at Echirolles, France. Each node has 2 x AMD rome 24 core (AMD EPYC 7402) with 128GB of RAM. The nodes are connected using the InfiniBand HDR interconnection. All the nodes have an RHEL 8.5 system. Our software stack is built with LLVM-15 and we used an Eviden OpenMPI implementation of MPI, built in its 4.1.5.2 version. For the experiments, we use RMA-Analyzer integrated in PARCOACH [7], a tool dedicated to the detection of deadlocks caused by an improper use

of collective calls in parallel applications. The performance analysis is done with MUST-RMA v1.9.0, available on github[1].

5.1 Implementation Details

Our contribution has been implemented in RMA-Analyzer, recently integrated in the tool PARCOACH, based on the LLVM compiler. An overview of the RMA-Analyzer framework is presented in [1]. It first collects all memory accesses that are contained within each epoch. To do so, relevant instructions such as memory accesses within an epoch, window creation and destruction (`MPI_Win_create`/ `MPI_Win_free`), epoch creation and destruction (`MPI_Win_lock_all`/ `MPI_Win_unlock_all`), and synchronization (`MPI_Win_flush_all`) functions are instrumented during the compilation phase. The instrumentation of the MPI functions are made using the PMPI (Profiling for MPI) interface. The LLVM alias analysis is used to reduce the number of Load/Store instrumentations.

During the execution, the MPI processes store each instrumented memory access in a BST and check for data races. If a data race is detected, RMA-Analyzer stops the execution of the program and reports the error with precise debug information (i.e., line of the conflicting instructions in the source code). More specifically, each time a remote access is initiated by a process (with `MPI_Put` or `MPI_Get`), an `MPI_Send` is called by RMA-Analyzer to inform the target process about this remote access. For each window, a thread is created to receive all the `MPI_Send`. At the end of the epoch (e.g., `MPI_Win_unlock_all`), all processes call `MPI_Reduce` in order to compute the number of remote accesses issued during the epoch towards its window and wait for the pending communications. The BST is implemented using the multiset containers provided by the C++ standard.

5.2 Method Validation

We developed a microbenchmark suite containing small programs written in C with correct and incorrect uses of MPI-RMA communications. This suite contains every combination of two one-sided operations by varying the order of the operations, the callers of the operations, and the location that will be accessed twice. The suite contains 154 codes in total and is composed of 47 codes containing a data race and 107 safe codes. Table 2 shows four programs from the suite. The names of the codes correspond to the combination used. For instance, the data race shown in Figure 2a is represented by the program named *ll_get_load_outwindow_origin_race*, in which an `MPI_Get` operation followed by a `Load` operation are issued by the same process (*ll* = *locallocal*), and both operations are accessing the same address space at *origin* side, outside its window (*origin_outwindow*). As shown in the table, all the tools are getting the right result for *ll_get_load_outwindow_origin_race* and *ll_get_get_inwindow_origin_safe*. However, a false negative is produced by MUST-RMA for *ll_get_load_inwindow_origin_race*. This is because ThreadSanitizer does not instrument stack arrays. Thus, MUST-RMA cannot consider them in its analysis. When using heap arrays, the error is detected by MUST-RMA. For *ll_load_get_inwindow_origin_safe*, a data race is detected by RMA-Analyzer because the implementation does not consider the order of instructions within a process. When a process makes a local memory access and then calls an RMA operation, no data race can occur. On the contrary, if the process first calls an RMA operation and then makes a local memory access, a data race can occur if both operations are accessing the same address and one of them is a write operation. Thus, RMA-Analyzer does not distinguish cases such as `Load-MPI_Get` and `MPI_Get-Load` and raises an error even though the first case does not contain any error. We fixed this problem in our errors detection algorithm.

Table 3 summarizes the results obtained with RMA-Analyzer, MUST-RMA and our contribution on our microbenchmark suite. All the false positives detected by RMA-Analyzer and the false negatives produced by MUST-RMA are caused by a lack of analysis precision and ThreadSanitizer instrumentation. Our contribution outperforms these tools with no false positive and no false negative.

To ensure our method is able to detect errors in bigger codes, we manually inserted a data race in MiniVite. We duplicated an `MPI_Put` operation as shown in Figure 9a. Then a data race may occur at target side since both `MPI_Put` operations are writting in the same address space of the target process. Both RMA-Analyzer and our contribution detect the data race. The output returned by our contribution is depicted in Figure 9b. The error message includes debug information like the name of the file and the line in the source code where the data race is detected.

```
for(int i = 0; i < num_comm_procs; i++){
  int target_rank = comm_proc[i];
  MPI_Put(scdata.data() + comm_proc_buf_disp[i], ssizes[
        target_rank], MPI_GRAPH_TYPE,
        target_rank, disp[target_rank], ssizes[target_rank],
              MPI_GRAPH_TYPE, commwin)
  MPI_Put(scdata.data() + comm_proc_buf_disp[i], ssizes[
        target_rank], MPI_GRAPH_TYPE,
        target_rank, disp[target_rank], ssizes[target_rank],
              MPI_GRAPH_TYPE, commwin)
}
```

(a) Code 3, extracted from MiniVite.

```
$ mpiexec -n2./miniVite -l -n100

Error when inserting memory access of type RMA_WRITE from file
./dspl.hpp:614 with already inserted interval of type RMA_WRITE
from file ./dspl.hpp:612. The program will be exiting now with
      MPI_Abort.
Error when inserting memory access of type RMA_WRITE from file
./dspl.hpp:614 with already inserted interval of type RMA_WRITE
from file ./dspl.hpp:612.The program will be exiting now with
      MPI_Abort.
```

(b) Report returned by our contribution on code 3.

Figure 9: Data race manually inserted in MiniVite and the output we returned to the developers.

5.3 Performance Analysis

In order to evaluate the overhead at runtime induced by the different approaches, we produced a performance analysis on two "real-life" applications: CFD-Proxy [20] and MiniVite [4]. As MPI-RMA programs are challenging to write, few MPI-RMA applications are available. CFD-Proxy is a proxy-application for computational fluid dynamics. The application uses the Passive Target synchronization mode - using `MPI_Win_lock_all` and `MPI_Win_unlock_all`

[1]https://github.com/RWTH-HPC/must-rma-correctness22-supplemental

Table 2: Comparison of RMA-Analyzer, MUST-RMA, and our contribution feedback on four codes from our microbenchmark suite (✓: error detected, x: no error found).

	RMA-Analyzer	MUST-RMA	Our Contribution
ll_get_load_outwindow_origin_race	✓	✓	✓
ll_get_get_inwindow_origin_safe	x	x	x
ll_get_load_inwindow_origin_race	✓	x	✓
ll_load_get_inwindow_origin_safe	✓	x	x

Table 3: Number of False Positive (FP), False Negative (FN), True Positive (TP) and True Negative (TN) reported by RMA-Analyzer, MUST-RMA, and our contribution on our microbenchmark suite.

	RMA-Analyzer	MUST-RMA	Our Contribution
FP	6	0	0
FN	0	15	0
TP	41	32	47
TN	107	107	107

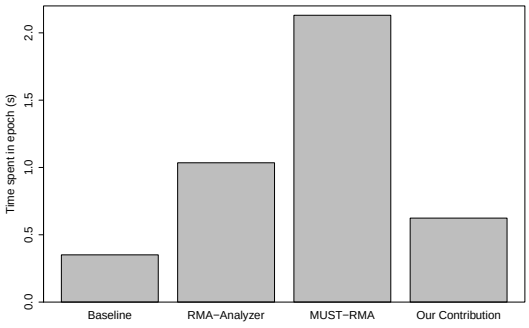


Figure 10: Cumulative time spent in the epochs in CFD-Proxy for each method (1 node, 12 ranks, 50 Iterations).

to create an epoch - and contains two windows per MPI process, and two epochs are called in the program: one per window.

Figure 10 presents a comparison of the cumulative time spent in the epochs for CFD-Proxy. This experiment has been executed on one node with 12 MPI processes. As illustrated, the overhead is greatly reduced with our contribution compared to RMA-Analyzer. In fact, thanks to the merging algorithm, the number of nodes in the BST is drastically reduced from a BST of size 90,004 with RMA-Analyzer to a BST of size 54 with our method (i.e a reduction of 99.94% of the number of nodes in the BST). This reduction of nodes is possible since the window allocated by a process is actually divided into the number of processes so all the other processes have a dedicated space in the window they can access. Thus, every remote accesses made by a process access the same address space: these accesses can then be merged into one node in the BST. As the execution time of the insertion and deletion operations in the BST depends on the size of the BST, a reduction of its size would

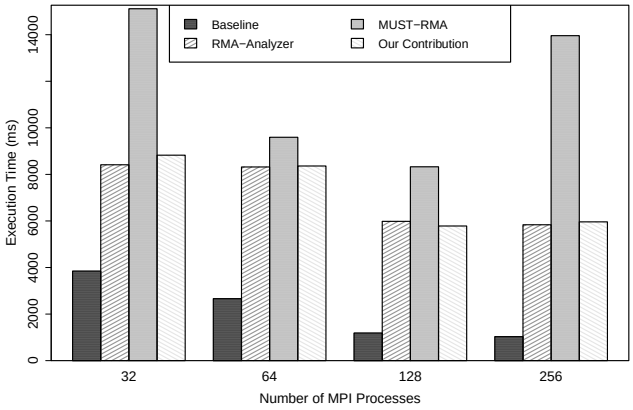


Figure 11: Execution-time of MiniVite with 32 to 256 MPI processes (2 to 16 nodes) and 640,000 vertices as input.

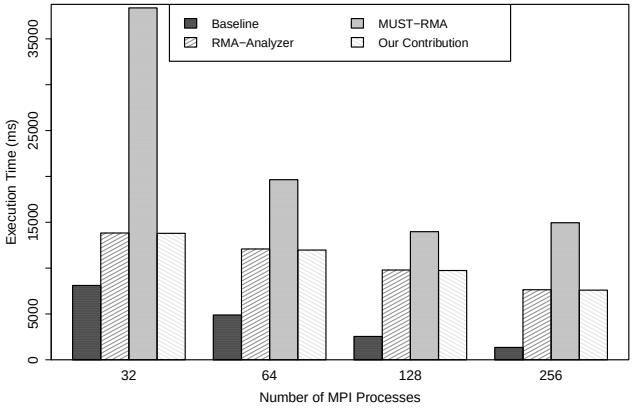


Figure 12: Execution-time of MiniVite with 32 to 256 MPI processes (2 to 16 nodes) and 1,280,000 vertices as input.

also reduce the execution time of the analysis. MUST-RMA has a significant slowdown as ThreadSanitizer instruments all memory accesses in the program while RMA-Analyzer and our contribution both use an alias analysis to filter out useless memory accesses for the error detection. Nonetheless, for our experiments, we only consider the execution time of the epoch.

Table 4: Number of Nodes in the BST for MiniVite for 32 to 256 MPI processes.

	RMA-Analyzer (640,000/1,280,000)	Our Contribution (640,000/1,280,000)	Reduction of Nodes
32	88,528/177,223	88,493/176,916	0.04%/0.17%
64	48,180/97,347	47,913/97,020	0.55%/0.34%
128	26,383/52,105	25,916/51,672	1.77%/0.83%
256	15,544/29,129	14,566/28,127	6.29%/3.44%

MiniVite is a proxy-application that implements a single phase of Louvain method in distributed memory for graph community detection. The application uses the Passive Target synchronization mode, and contains only one epoch. Figure 11 presents a comparison of the time spent in the epoch when running MiniVite from 32 on 2 nodes to 256 processes on 16 nodes When comparing RMA-Analyzer with our contribution, the performance is substantially the same since our contribution does not merge many nodes in the BST. As shown in Table 4, the number of nodes in the BST is reduced by less than 4%. The cost induced by the new insertion algorithm is thus not compensated by the reduction of time spent in the operations on the BST (insertion, remove and search). This low-level of node merging is caused by memory accesses on attributes of adjacent objects. However, the memory space of these attributes are not adjacent to one another. Thus, the nodes associated to these attributes memory accesses cannot be merged. It should be mentioned that when the number of processes is increased, the number of communications between processes increases while the workload per process is reduced. Thus, the overlap between communications and computations is less important with more processes. This explains why the gains in the execution time is not really visible between 128 and 256 processes even for the baseline execution. Thus, when increasing the problem size as in Figure 12 which shows the execution time with 1,280,000 vertices as input, a gain can be seen when running with 256 processes compared to a run with 128 processes because each process has more work to do and a better communication/computation overlap is possible. Nevertheless, the overhead implied by MUST-RMA is even more important when running on a large number of processes. Indeed, in addition to the slowdown introduced by the over-instrumentation of ThreadSanitizer as for CFD-Proxy, when the number of processes greatly increases, the size of the vector clock that is sent to other processes also increases. Thus, sending larger messages also adds overhead at runtime.

6 DISCUSSION

In this section, we first discuss some synchronization solutions within an epoch in order to avoid a data race by using the `MPI_Barrier` and `MPI_Win_flush_all` functions (1). We then discuss the problem of `MPI_Win_flush` instrumentation in RMA-Analyzer (2). Finally, we address the lack of performance with MiniVite due to non-adjacent accesses (3).

(1). According to the MPI standard, an `MPI_Barrier` which waits for all the processes of the communicator actually does not terminate the communications. However, in practice, `MPI_Barrier` terminates communications in most MPI implementations. In our approach, we decided to meet the standard. Using only `MPI_Win_flush_all` instead does not fix the problem since it does not ensure all processes have finished the communications. The solution consisting in calling `MPI_Barrier` after `MPI_Win_flush_all` should be preferred to synchronize the communications within an epoch.

(2). `MPI_Win_flush` is a synchronization function within an epoch that *"completes all outstanding RMA operations initiated by the calling process to the target rank on the specified window. The operations are completed both at the origin and at the target"* [13]. However, instrumenting `MPI_Win_flush` is not trivial since the latter only guarantees the ordering of communications of the calling process. Thus, the target of `MPI_Win_flush` is not aware of the synchronization call issued by the origin process and cannot know in which order remote accesses from several other processes toward its window will complete. That is why simply cleaning the BST of the process calling `MPI_Win_flush` may lead to false negatives. For instance, a false positive was detected by RMA-Analyzer and MUST-RMA when running them on CFD-Proxy because `MPI_Win_flush` is not well instrumented by the tools. As a consequence, we cannot support this synchronization function yet. We let for a future work a more in-depth study on the handling of the instrumentation of `MPI_Win_flush`.

(3). As described in Section 5.3, when runnning on MiniVite, there is no significant gain using our contribution compared to RMA-Analyzer because memory accesses are not adjacent. Some studies address this problem by using polyhedra to abstract memory regions and thus enable the compression of memory accesses even if they are not adjacent (e.g., [9]). We thus assume that using these concepts, the merging algorithm can be extended to non-adjacent accesses when we can ensure that no accesses will be done between the accesses.

7 CONCLUSION

This paper proposes a better data race detection technique for the tool RMA-Analyzer [1]. We present a way to better manage the BST that is used to store all the accesses to the address space of an MPI process. We introduced a new insertion algorithm so the resulting BST detects more errors than the previous version of RMA-Analyzer. For future works, we plan to enhance the static analysis proposed by Saillard *et al.* [16] to detect more errors at compile time. We also plan to combine this static analysis to RMA-Analyzer in order to reduce the overhead at runtime.

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 2021 - European MPI Users's Group Meeting*. Munich, Germany. https://doi.org/10.1145/1122445.1122456

[2] 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 *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 499–510. https://doi.org/10.1109/SC.2014.46

[3] Thanh-Dang Diep, Karl Fürlinger, and Nam Thoai. 2018. MC-CChecker: A Clock-Based Approach to Detect Memory Consistency Errors in MPI One-Sided Applications. In *Proceedings of the 25th European MPI Users' Group Meeting* (Barcelona, Spain) *(EuroMPI'18)*. Association for Computing Machinery, New York, NY, USA, Article 9, 11 pages. https://doi.org/10.1145/3236367.3236369

[4] Sayan Ghosh, Mahantesh Halappanavar, Antonino Tumeo, Ananth Kalyanaraman, Hao Lu, Daniel Chavarrià-Miranda, Arif Khan, and Assefaw Gebremedhin. 2018. Distributed Louvain Algorithm for Graph Community Detection. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 885–895. https://doi.org/10.1109/IPDPS.2018.00098

[5] Tobias Hilbrich, Martin Schulz, Bronis R. de Supinski, and Matthias S. Müller. 2010. MUST: A Scalable Approach to Runtime Error Detection in MPI Programs. In *Tools for High Performance Computing 2009*, Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 53–66.

[6] Torsten Hoefler, James Dinan, Rajeev Thakur, Brian Barrett, Pavan Balaji, William Gropp, and Keith Underwood. 2015. Remote Memory Access Programming in MPI-3. *ACM Trans. Parallel Comput.* 2, 2, Article 9 (jun 2015), 26 pages. https://doi.org/10.1145/2780584

[7] Pierre Huchant, Emmanuelle Saillard, Denis Barthou, and Patrick Carribault. 2019. Multi-Valued Expression Analysis for Collective Checking. In *EuroPar*. Göttingen, Germany. https://hal.science/hal-02390025

[8] Ali Jannesari, Kaibin Bao, Victor Pankratius, and Walter Tichy. 2009. Helgrind+: An efficient dynamic race detector. 1–13. https://doi.org/10.1109/IPDPS.2009.5160998

[9] Alain Ketterlin and Philippe Clauss. 2008. Prediction and trace compression of data access addresses through nested loop recognition. In *6th annual IEEE/ACM international symposium on Code generation and optimization*. ACM, Boston, United States, 94–103. https://doi.org/10.1145/1356058.1356071

[10] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (jul 1978), 558–565. https://doi.org/10.1145/359545.359563

[11] Mingzhe Li, Xiaoyi Lu, Khaled Hamidouche, Jie Zhang, and Dhabaleswar K. Panda. 2016. Mizan-RMA: Accelerating Mizan Graph Processing Framework with MPI RMA. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*. 42–51. https://doi.org/10.1109/HiPC.2016.015

[12] Mingzhe Li, Xiaoyi Lu, Sreeram Potluri, Khaled Hamidouche, Jithin Jose, Karen A. Tomko, and Dhabaleswar Kumar Panda. 2014. Scalable Graph500 design with MPI-3 RMA. *2014 IEEE International Conference on Cluster Computing (CLUSTER)* (2014), 230–238.

[13] Message Passing Interface Forum. 2021. *MPI: A Message-Passing Interface Standard Version 4.0*. https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

[14] Arndt Müehlenfeld and Franz Wotawa. 2007. Fault Detection in Multi-Threaded C++ Server Applications. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Jose, California, USA) *(PPoPP '07)*. Association for Computing Machinery, New York, NY, USA, 142–143. https://doi.org/10.1145/1229428.1229457

[15] Mi-Young Park and Sang-Hwa Chung. 2009. Detecting Race Conditions in One-Sided Communication of MPI Programs. In *2009 Eighth IEEE/ACIS International Conference on Computer and Information Science*. 867–872. https://doi.org/10.1109/ICIS.2009.170

[16] Emmanuelle Saillard, Marc Sergent, Tassadit Célia Aitkaci, and Denis Barthou. 2022. Static Local Concurrency Errors Detection in MPI-RMA Programs. In *Correctness 2022 - Sixth International Workshop on Software Correctness for HPC Applications*. Dallas, United States. https://hal.inria.fr/hal-03882459

[17] 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 *2022 IEEE/ACM Sixth International Workshop on Software Correctness for HPC Applications (Correctness)*. 27–36. https://doi.org/10.1109/Correctness56720.2022.00009

[18] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications* (New York, New York, USA) *(WBIA '09)*. Association for Computing Machinery, New York, NY, USA, 62–71. https://doi.org/10.1145/1791194.1791203

[19] Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov. 2012. Dynamic Race Detection with LLVM Compiler. In *Runtime Verification*, Sarfraz Khurshid and Koushik Sen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 110–114.

[20] Christian Simmendinger. 2014. PGAS Community Benchmarks CFD-Proxy version 1.0.1. https://github.com/PGAS-community-benchmarks/CFD-Proxy.