Static-Dynamic Analysis for Performance and Accuracy of Data Race Detection in MPI One-Sided Programs

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

Static-Dynamic analysis for Performance and Accuracy of Data Race Detection in MPI One-Sided Programs

Radjasouria Vinayagame[1], Van Man Nguyen[1], Marc Sergent[1], Samuel Thibault[2], and Emmanuelle Saillard[3]

[1] Eviden, Echirolles, France
[2] University of Bordeaux, Bordeaux, France
[3] Inria, Bordeaux, France

Abstract. To take advantage of asynchronous communication mechanisms provided by the recent platforms, the Message Passing Interface (MPI) proposes operations based on one-sided communications. These operations enable a better overlap of communications with computations. However, programmers must manage data consistency and synchronization to avoid data races, which may be a daunting task. In this paper, we propose three solutions to improve the performance and the accuracy of the data race detection in MPI one-sided programs. First, we extend the node-merging algorithm based of a Binary Search Tree (BST) presented in a previous work that keeps track of memory accesses during execution to take into account non-adjacent memory accesses. Then, we use an alias analysis to reduce the number of load/store instrumented. Finally, we extend our analyses to manage synchronization routines. Our solutions have been implemented in PARCOACH, a MPI verification tool. Experiments on real-life applications show that our contributions lead to a better accuracy, a reduction of the memory usage by a factor up to 4 of the dynamic analysis and a reduction of the overhead at runtime at larger scale.

Keywords: MPI One-Sided Communications, Verification, Data Race, Non-adjacent Memory Accesses.

1 Introduction

The Message Passing Interface (MPI) standard provides one-sided communications, also known as Remote Memory Access (RMA). It enables a rank to remotely access and manipulate memory located on a target rank without requiring an explicit coordination from the latter. In comparison to traditional MPI point-to-point two-sided communication, such as `MPI_Send` and `MPI_Recv`, one-sided communications decouple data transfers and synchronizations. Therefore, MPI one-sided communications have shown good performance in several applications like in [2] where Ghosh et al. compare the performance of a program that implements an approximate weighted graph matching algorithm when using one-sided and two-sided communications. As one-sided communications enable direct memory access and reduce communication overhead, their use can improve the performance of applications, especially those with irregular communication pattern.

Nonetheless, developers still use two-sided communications in their programs because one-sided communications are difficult to use. Indeed, with one-sided communications, developers are exposed to data races if they do not ensure memory consistency, which can be a challenging task. Some approaches exist to detect data races in MPI one-sided applications and help developers write correct and efficient programs. However, these approaches come with restrictions (they do not consider all

the features presented in the MPI standard) and most of them imply a significant overhead at runtime. This paper proposes new methods to improve the performance and the accuracy of the data race detection in PARCOACH. We improve the work presented in [10] with the following contributions:

- A new algorithm to merge non-adjacent accesses in the Binary Search Tree (BST) during the execution of programs
- The use of an alias analysis to reduce load/store instrumentations at compile time
- Considering flush synchronizations in the data race detection algorithm to avoid false positives

The paper is organized as follows: Section 2 provides background elements, the key concepts this work relies on and related work. Section 3 describes our contributions to enhance the existing PARCOACH static and on-the-fly data race detection analyses in order to improve the accuracy, and reduce the memory usage during execution. Section 4 shows results on two applications and compares our contributions against the previous version of PARCOACH and MUST-RMA, the only other active state-of-the-art tool that can detect data races in MPI one-sided programs. Finally, Section 5 concludes this work.

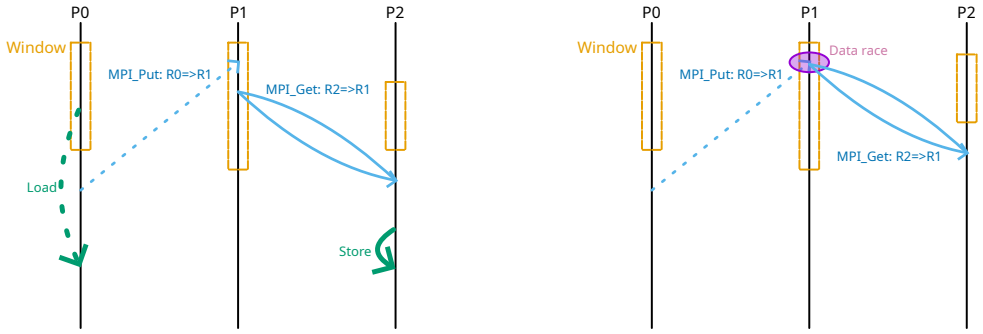
2 Background and Related Work

2.1 MPI-RMA

When using MPI one-sided communications, each process exposes a *distributed shared memory* that can be accessed by all MPI processes. These memory regions are called *windows*. To perform remote memory accesses to these windows, developers must define an *epoch*. Within an epoch, MPI-RMA proposes several communication operations which involve two processes: *origin* and *target*. Process *origin* issues the MPI-RMA communication while the *target* process window is accessed via the communication. In this paper, we only consider programs that use `MPI_Fence` and `MPI_(un)lock_all` to create an epoch. We support the two major one-sided communication operations: `MPI_Put` and `MPI_Get`. Figure 1 shows examples of these operations. In both subfigures, `MPI_Put` writes a value owned by the *origin* process R0 to the window of the *target* process R1, and `MPI_Get` allows the *origin* process R2 to locally retrieve a value from the window of the *target* process R1. It should be mentioned that a process can also access its own window through local memory accesses (`LOAD` and `STORE`).

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. In such programs, it is the developer's responsibility to ensure memory consistency to avoid data races. A data race occurs in a MPI-RMA program if (1) two operations access the same data, (2) at least one operation is a one-sided communication operation, and (3) at least one operation is a WRITE operation. Figure 1b shows an example of a data race. Both processes R0 and R2 access the same memory space of R1 (in purple on the figure) with one-sided operations including the Put operation that writes on the memory space. The value read by the Get operation is thus undefined.

To avoid data races and ensure the completion of communications within an epoch, the MPI-3 standard proposes sychronization routines such as `MPI_Win_flush` and `MPI_Win_flush_all` that can only be used with the passive target synchronization mode. According to the MPI standard, `MPI_Win_flush` 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



(a) Correct scenario (b) Incorrect scenario

Fig. 1: Examples of correct and incorrect use of Put/Get operations with three MPI processes. From the *origin* process perspective, plain lines represent $WRITE$ operations and dashed lines represent $READ$ operations. Thin and thick lines respectively represent remote and local operations.

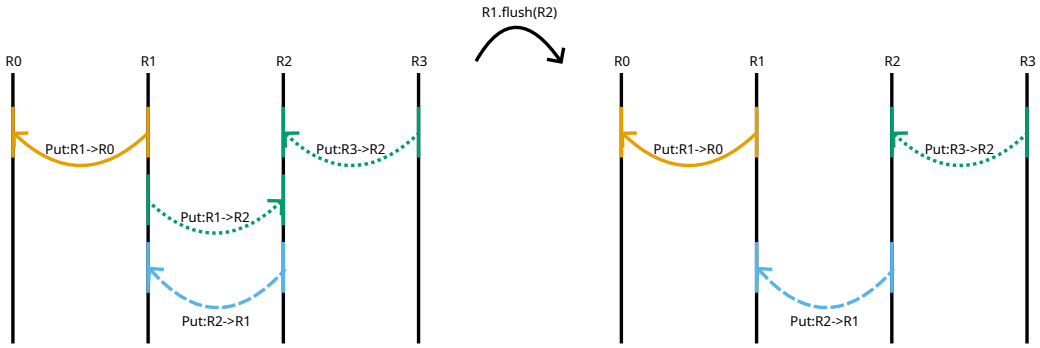


Fig. 2: Behavior of the `MPI_Win_flush` routine. The coloured lines represent ongoing communications. The line type characterizes the target of the communication. The call to `MPI_Win_flush` by R1 toward R2 completes all the communications initiated by R1 for which R2 is the target.

target. An example of how the routine works is depicted in Figure 2. Given four ranks initiating communications toward other ranks, when R1 calls `MPI_Win_flush` on R2, only communications initiated by R1 and targeted to R2 are completed after the function returns. Completion of communications from R2 to R1 is however not guaranteed. It is crucial to consider these synchronizations when looking for data races.

2.2 Related Work

Several approaches exist to detect data races in MPI one-sided programs. In [5], Park et al. developed an approach that creates a mirror window that stores all one-sided communications. Upon execution of a communication, the tool checks if a data race can occur with a previous communication in the mirror window. This approach does not consider local memory accesses and can miss errors. *MC-CChecker* [1] uses a post-mortem analysis based on the encoded vector clock to detect data races.

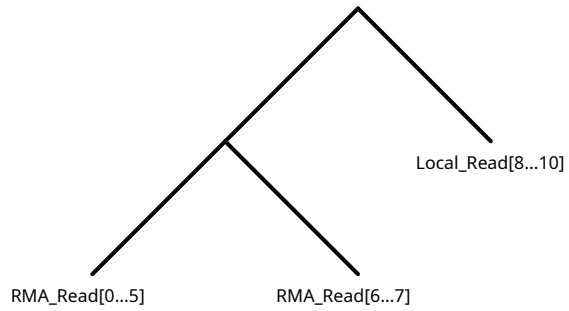


Fig. 3: Example of a Binary Search Tree in PARCOACH.

It is only compatible with the MPI-2 standard and thus does not support newer MPI one-sided features such as the `MPI_Win_lock_all`/`MPI_Win_unlock_all` functions.

To the best of our knowledge, MUST-RMA [7] and PARCOACH [11, 6, 10] are the only two active tools capable of detecting errors in MPI-RMA programs. MUST-RMA combines MUST [4], a dynamic MPI verification tool and ThreadSanitizer, a shared-memory data race detector [8]. It constructs concurrent regions based on the happens-before relation and forwards them to ThreadSanitizer that checks for data races.

PARCOACH combines static and dynamic analyses to detect data races caused by one-sided communications. The static analysis performs a traversal of the Control Flow Graph to report errors at origin side and instrument memory accesses. The dynamic analysis builds a Binary Search Tree (BST) of memory addresses during execution. A node in the BST is called an *Access*. It represents one memory access and contains the following relevant information about it:

- An interval representing the limits of the memory access. All accessed addresses are contained in this interval.
- The access type: local or remote ($Local\_*$ or RMA_*) and read or write operation ($*\_READ$ or $*\_WRITE$). Note that a one-sided operation implies two memory accesses: one for origin rank and one for the targeted rank.
- Debug information for error reporting (e.g., line in the source code).

In the BST, nodes are ordered by the lower bound of the memory adresses. An example of a BST containing three nodes is shown in Figure 3.

In [10], Vinayagame et al. enhance this approach with a node-split algorithm that splits nodes of the BST to ensure no intervals overlap and a node-merging algorithm that reduces the size of the BST. The resulting algorithm that insert a node in the BST is presented in Algorithm 1. This algorithm first checks if a new access $NewAcc$ can lead to a data race with a previous memory access by identifying the conflicting intersections of memory access intervals (line 2). If no data race is possible, a BST node representing this new access is created. The algorithm identifies the intervals intersecting with $NewAcc$ (line 4). These intersections do not cause a data race because they have compatible access types. Based on these intersections, new nodes are created, called fragments, that are not overlapping (line 5). All of these nodes are then merged when possible and especially when they represent adjacent addresses (line 6). Finally, the merged nodes replace the previous nodes (line 7). For example, in Figure 3, nodes $RMA\_READ[0...5]$ and $RMA\_READ[6...7]$ can be merged, and replaced by the node $RMA\_READ[0...7]$. Despite good results on different benchmarks, this

Algorithm 1 Insertion of a memory access in the BST

```
1: function INSERT_BST(NewAcc, BST)
2:     HasError ← dataRaceDetection(NewAcc, BST)        ▷ report an error in case of a data race
3:     if !HasError then
4:         InterAcc ← get_intersecting_accesses(NewAcc, BST)
5:         FragAcc ← create_accesses(InterAcc, NewAcc)
6:         MergedAcc ← merge_accesses(FragAcc)
7:         finish_insertion(InterAcc, MergedAcc, BST)
```

approach has several limitations. First, non-adjacent memory accesses are not considered in the node-merging algorithm. This limits the speedup the analysis could achieve on such applications like *MiniVite* [3] that are making equidistant memory accesses. Additionally, PARCOACH does not consider synchronizations with `flush` operations in its analyses. The tool can then report false positives. Indeed, in the CFD-Proxy [9] application, PARCOACH returns an error because of an uncaught `MPI_Win_flush`.

In this paper, we propose an extension of the node-merging algorithm that takes into account non-adjacent memory accesses to reduce the size of the BST. We also improve the memory accesses instrumentation described in [6]. This also reduces the number of nodes in the BST. Finally, we extend the dynamic analysis presented in [10] to support `flush` synchronizations during the detection of data races to avoid false positives.

3 Contributions

In this section, we present three contributions that aim at reducing the memory usage of the analysis and at improving its accuracy. Section 3.1 explains how equidistant memory accesses can be merged in the BST to reduce the number of nodes. Section 3.2 proposes a solution to reduce the number of instrumentations to relieve the dynamic analysis. Finally, Section 3.3 presents a way to consider synchronizations in the analysis of data races to increase its accuracy.

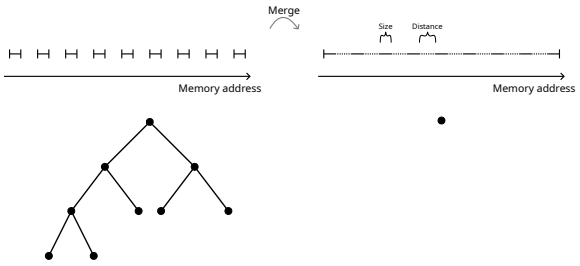
3.1 Merging Non-Adjacent Accesses in the BST

The goal of this section is to reduce the size of the BST by merging nodes in the BST that represent equidistant memory accesses. Thus, as shown in Figure 4b, instead of storing nine nodes in the BST, only one, representing the nine memory accesses, should be inserted. Thus, the BST induced by the program presented in Figure 4a should contain one node instead of the number of iterations as it was the case with PARCOACH.

In order to take into account non-adjacent memory accesses, we propose a new representation of a node that now describes equidistant sub-intervals instead of a single interval. To this end, two new pieces of information are added in the *Access* structure: a *Size* attribute representing the size of each sub-interval, and a *Distance* attribute representing the constant distance of the sub-intervals. These new attributes are represented in Figure 4b. Nonetheless, the analysis still has to ensure that no sub-intervals are intersecting in the BST. Additionally, the analysis should represent the memory access issued in a code such as the one presented in Figure 5a as two nodes represented in Figure 5b. As a consequence, the functions of the insertion algorithm presented in Algorithm 1 have been amended to improve the analysis. These changes are explained in the following paragraphs.

```
struct data A {
  char buffer1[MAX_BUFFER_SIZE];
  char buffer2[MAX_BUFFER_SIZE];
}
struct data A[MAX_TAB_SIZE];
for(int i = 0; i < MAX_TAB_SIZE; i++)
  MPI_Put(A[i].buffer2,MAX_BUFFER_SIZE,1,...)
```

(a) Example of code with non-adjacent memory accesses.

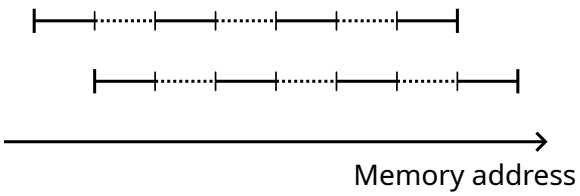


(b) Merging of equidistant memory accesses in the BST.

Fig. 4: Example of a code making non-adjacent memory accesses and how the latter should be represented in the BST.

```
struct data {
  char buffer1[MAX_BUFFER_SIZE];
  char buffer2[MAX_BUFFER_SIZE];
}
struct data A[MAX_TAB_SIZE];
for(int i = 0; i < MAX_TAB_SIZE; i++)
  MPI_Put(A[i].buffer1,MAX_BUFFER_SIZE,1,...)
[some computations]
for(int i = 0; i < MAX_TAB_SIZE; i++)
  MPI_Put(A[i].buffer2,MAX_BUFFER_SIZE,1,...)
```

(a) Example of code with equidistant memory accesses that are interlacing.



(b) Merging the memory accesses into two nodes

Fig. 5: Example of desired representation of interlaced non-adjacent accesses.

`get_intersecting_intervals` The purpose of this function is to find all nodes intersecting with $NewAcc$. It now considers two extra non-intersecting nodes on the right of the rightmost inserted node, and two extra nodes on the left of the leftmost inserted node. These extra nodes may be useful for the merging of the nodes in the case where they can be merged with the new created node. Additionally, in order to not miss any intersecting interval, this function must check if the nodes that contain the smallest and largest intervals inserted so far are intersecting with $NewAcc$. An example of its execution is shown in Figure 6. We can distinguish four types of nodes with `get_intersecting_intervals`:

- Nodes intersecting with $NewAcc$ returned by the function during the traversal of the BST. These node are referred as "*Intersecting*", and are the only nodes reported by the original `get_intersecting_intervals` of PARCOACH.
- Nodes that are not intersecting $NewAcc$ but are returned because they might be mergeable. These nodes are tagged "*ExtraTwo*".
- Nodes intersecting with $NewAcc$ but not returned during the traversal because a non-intersecting node separates them from the rest of the intersecting nodes. These nodes are referred as "*Included in*". These nodes (which can be arbitrarily far in the BST) will be reached from a returned node through interval-inclusion pointers represented as vertical arrows tagged "*Is included in*" (maintained hierarchically).

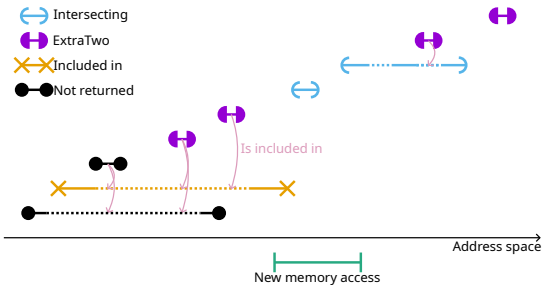


Fig. 6: Determining which intervals are intersecting with the new access.

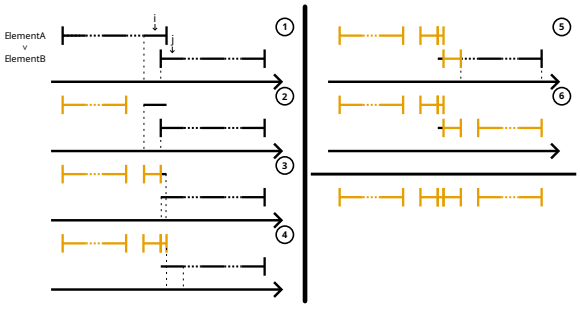


Fig. 7: Fragmentation process of two intersecting intervals.

– Nodes that are not returned by the function, which are called "*Not returned*" in the Figure 6.

`create_accesses` Consider a set X of *Access* objects, that are now composed of several sub-intervals, and the memory access $NewAcc$ that is being inserted in the BST. If $NewAcc$ intersects with a sub-interval of any element of X, then this function creates new *Access* objects representing disjoint memory accesses. For each memory access $m \in X$, if $NewAcc$ is intersecting with at least one sub-interval of m, then both m and $NewAcc$ are divided into fragments. An example of how the fragments are computed is depicted in Figure 7. Given nodes $ElementA \in X$ and $ElementB \in X$ such as there is a sub-interval of $ElementA$ that intersects with a sub-interval of $ElementB$, step ① gets the two aforementioned sub-intervals. Step ② then creates a node representing the first sub-intervals that are not intersecting with the sub-intervals of the other node. Steps ③, ④, and ⑤ create disjoint fragment nodes until the function reaches the end of one of the nodes. Finally, step ⑥ tries to create a node representing the remaining sub-intervals.

`merge_accesses` Given a set of ordered and disjoint memory accesses, this function merges the elements of this set when possible. It looks per batch of three nodes if they can be merged. They have to be equidistant, with the same debug information, and have the same access type and size. If the batch of three nodes can be merged, the function looks for the next nodes in the set and adds them in the merged node. If the three nodes cannot be merged, the function creates one or two nodes depending on whether the first two nodes can be merged.

3.2 Reducing Memory Accesses Instrumentations for the Dynamic Analysis

The original dynamic analysis is implemented by instrumenting, at compile time, all memory accesses within an epoch, whether they are local or remote accesses. This is concerning since the instrumentation of those instructions adds an overhead to the execution time of the analysis, which slows down the execution of the whole application.

The goal of this section is to reduce the number of instrumentations for the dynamic analysis by removing at compile time the instrumentation of operations that are not subject to data races.

Table 1 summarizes all the possible combinations of operations, with the colored cells representing the ones subject to data races. Depending on which ranks are initiating the operations and toward which rank, we discern three types of data race:

R0 (Origin)	R1 (Target)
Win_lock_all	Win_lock_all
Get(**buf**, **1**, **X**)	
printf(**buf**)	
Win_unlock_all	Win_unlock_all

(a) Local concurrency errors detected by the static analysis.

R0 (Origin)	R1 (Target)
Win_lock_all	Win_lock_all
Put(**buf**, **1**, **X**)	**X** = **0**
Win_unlock_all	Win_unlock_all

(b) *target* rank accesses its window while a communication has been initiated at it.

R0 (Origin 1)	R1 (Target)	R2 (Origin 2)
Win_lock_all	Win_lock_all	Win_lock_all
Put(**buf**, **1**, **X**)		**Put**(**buf**, **1**, **X**)
Win_unlock_all	Win_unlock_all	Win_unlock_all

(c) Two ranks are making a remote access toward the same target rank.

R0 (Origin)	R1 (Target)
Win_lock_all	Win_lock_all
Get(**buf**, **1**, **X**)	
Flush(**1**)	
printf(**buf**)	
Win_unlock_all	Win_unlock_all

(d) `Flush` makes the operations safe.

Fig. 8: Examples of the different types of data races. **X** represents the window of R1.

- *Local-Local* data races, also known as local concurrency errors, involve two operations that are called by the same rank. Table 1a presents combinations of operations that can lead to this type of data race and Figure 8a presents an example of this type of data race.
- *Remote-Local* data races may occur when an origin rank initiates a communication toward a target rank while the latter also makes an operation. Table 1b describes these cases and Figure 8b presents one of them.
- *Remote-Remote* data races are induced by two origin ranks operating a remote access toward the same target rank. Table 1b depicts these cases and Figure 8c highlights one of them.

The static analysis presented by Saillard et al. [6] enables the detection of local concurrency errors. Thus, the analysis can detect the data races presented in blue and green cells in Table 1a. Remote-Local data races can only occur in the window of a rank: an origin performs a remote access to the window of the target rank while the latter also accesses its own window. Remote-Remote data races cannot be predicted at compile time, especially when the target of the communication is not known at compile time, which is the case in most applications. That is why all remote operations have to be instrumented.

By combining these three observations, we present a new instrumentation algorithm that instruments operations in three cases:

- MPI One-Sided Communications. This is done by instrumenting all `MPI_Put` and `MPI_Get` instructions.
- Local concurrency errors. This is done by first performing the static analysis proposed in [6] and then instrumenting the instructions flagged by the static analysis.
- Load/Store instructions that access the window. To that end, an alias analysis is used to check if a Load/Store operation is accessing a part of the window.

It should be mentioned that in the cases that are in the scope of the static analysis, there is no false negative. Indeed, the pointer analysis used in the approach is conservative, and flags any

	Get	Put	Load	Store
Get				
Put			x	
Load	x	x	x	x
Store	x	x	x	x

(a) Local-Local table: an origin rank first calls an instruction given in row and a second instruction given in column.

	Get	Put	Load	Store
Get			x	
Put				
Load	x		x	x
Store			x	x

(b) Remote-Local table: an origin rank calls an instruction given in row and a target calls an instruc- tion given in column.

	Get	Put
Get		
Put		

(c) Remote-Remote table: two origin ranks are remotely accessing the window of the same target rank.

Table 1: Pair of instructions that are instrumented. X cells are not instrumented because they are not subject to data race. Hatched cells (in grey) are instrumented because they involve two remote operations. Dotted cells (in blue) are instrumented if a data race is reported by the static analysis. Filled cells (in red) are instrumented if the local memory access accesses the window of a rank.

undecidable situations as a danger that has to be checked at runtime. As a consequence, we are not missing any instrumentation of operations that might lead to a data race.

Thereby, the new instrumentation algorithm instruments all the relevant instructions for the dynamic data race analysis while ensuring to not instrument instructions marked as X in Table 1 that represent safe memory accesses.

3.3 Considering Synchronizations in the Analysis

In this section, we present a solution to support synchronization routines such as `MPI_Win_flush` and `MPI_Win_flush_all`. Being able to properly analyse the behavior of codes using these calls improves the accuracy of data race detection tools.

MPI one-sided programs decouple data movements and synchronizations. Indeed, to allow a process the reuse of data involved in a remote operation, synchronization functions such as `MPI_Win_flush` can be called to ensure that a remote operation is completed. For example, to resolve the data race depicted in Figure 8a, a `MPI_Win_flush` can be inserted between the two accesses to buf as shown in Figure 8d. This synchronization ensures that the buf variable can be reused by the second access, since it waits for the completion of the communication.

To instrument the `MPI_Win_flush` routine, in addition to the BST of *Access* objects, a 2D communication array is created for each MPI process to store all the communications in which it is involved. This implies that each time a remote operation is initiated, the *target* MPI process is also notified so the latter can update its communication array. Thus, when `MPI_Win_flush` is called by a rank A for a rank B, A looks in its communication array for communications for which A is the origin and where B is the target. The nodes in its BST that are associated with this communication, which are the related memory accesses, are then removed. The same process is made for rank B. As

a consequence, any new memory access that would have been conflicting with a node in the BST, without the `MPI_Win_flush` operation, will not raise a data race since the aforementioned node is not in the BST anymore.

4 Experimental Results

In this section, we compare the impact of the different contributions presented in this paper to the two state-the-art approaches that are MUST-RMA [7] and PARCOACH [10]. We compare these approaches on two-real life applications: CFD-Proxy [9] and Mini-Vite [3]. We performed our experiments on an Eviden cluster that belongs to the Eviden R&D department, located at Echirolles, France. Each node has 2 x AMD ome 24 core (AMD EPYC 7402) with 128GB of RAM. The nodes are connected using the InfiniBand HDR interconnection. All the nodes run an RHEL 8.8 system. Our software stack is built with LLVM-15 and we used an Eviden fork of OpenMPI, in version 4.1.6.

4.1 Implementation Details

Our contributions have been implemented on top of the analyses of PARCOACH [4], which is itself based on the LLVM compiler. Relevant instructions are instrumented at compile time. The BST is implemented using the *multiset* containers provided by the C++ standard.

In the following, in addition to the base version of the applications which is running without any analysis (referred as *None*), we compare the performance of five combinations:

- *PARCOACH* is the base version of the application using the algorithm presented in [10]. This version can be found on the commit tagged `Merge-adjacent`.
- *PARCOACH+Merge* represents the implementation of the new node-merging algorithm presented in Section 3.1 (commit tagged `Merge-non-adjacent`).
- *PARCOACH+Instr* is the implementation of the reduction of instrumentations solution presented in Section 3.2 (commit tagged `Instru`).
- *PARCOACH+Both* gathers the two previous implementations (commit tagged `Merge-non-adjacent+Instru`).
- *PARCOACH+Flush* is the implementation considering the synchronizations as presented in Section 3.3 (commit tagged `Flush`).
- *MUST-RMA* is the state-of-the-art approach for data race detection in MPI-RMA programs presented in Section 2.2. The performance analysis is done with MUST-RMA v1.9.0 [5]

4.2 Method Validation

We added 18 test codes using `MPI_Win_flush` to the PARCOACH test suite in order to illustrate the accuracy of our approach. This new test suite includes the code presented in 8d. *PARCOACH+Flush* does not report a data race anymore while the previous version of *PARCOACH* reports a false positive.

We run the different approaches on CFD-Proxy [9] a proxy-application for computational fluid dynamics. In the application, each rank operates two `MPI_Put` communications that are separated by a `MPI_Win_flush` synchronization. The application is correct and *PARCOACH+Flush* is the only approach that does not detect a data race between the two `MPI_Put` operations.

[4] https://gitlab.inria.fr/parcoach/parcoach
[5] https://github.com/RWTH-HPC/must-rma-correctness22-supplemental

	# Load+Store instr. (/total)
PARCOACH	20 (/102)
PARCOACH+Merge	20 (/102)
PARCOACH+Instr	10 (/59)
PARCOACH+Both	10 (/59)

Table 2: Number of Load/Store instrumentations for Mini-Vite

Number of procs	32	64	128	256
PARCOACH	176,460	96,762	51,691	28,671
PARCOACH+Merge	40,531	23,006	12,081	7,041
PARCOACH+Instr	176,460	96,523	51,247	27,785
PARCOACH+Both	40,503	22,815	11,993	6,980

Table 3: Number of nodes in the BST when running on MiniVite with a problem size of 1,280,000, depending on the number of processes (in column)

Number of procs	32	64	128	256
PARCOACH	11.29	6.19	3.31	1.84
PARCOACH+Merge	3.24	1.84	0.97	0.56
PARCOACH+Instr	11.29	6.18	3.28	1.78
PARCOACH+Both	3.24	1.83	0.96	0.56

Table 4: Size in MB of the BST when running on MiniVite with a problem size of 1,280,000, depending on the number of processes (in column)

4.3 Performance Analysis

To evaluate the overhead introduced by the different approches, this section presents a performance analysis on MiniVite [3] which is a proxy-application that implements a single phase of Louvain method for graph community detection.

Table 2 summarizes the number of `LOAD`/`STORE` instructions instrumented. As expected, the *PARCOACH+Instr* and *PARCOACH+Both* approaches reduce the number of instrumentations by a factor up to 2. It is noteworthy that the approach instruments less files because it considers that the instructions in some files are not relevant for instrumentation, because it was able to ascertain that no data race is possible.

Table 3 gives the number of nodes in the BST when running the different PARCOACH approaches on Mini-Vite with a problem size of 1,280,000. On the one hand, the number of nodes is reduced by the *PARCOACH-Instr* approach since it instruments less memory accesses. The reduction of nodes is small because the reduction of instructions did not remove instructions called several times and also because the alias analysis used for the approach is conservative and wrongly considers that a lot of memory accesses are aliasing the window. On the other hand, *PARCOACH-Merge* considerably reduces the number of nodes by a factor up to 4. This is thanks to the equidistant memory accesses that are merged into one node in the BST. Finally, *PARCOACH-Both* is the best approach in terms of memory usage since it reduces the number of instrumentations and merges the remaining memory accesses. Figure 4 shows the size used by the BST in memory. Since each node stores more information in *PARCOACH-merge* and *PARCOACH-Both*, the size of a single node is increased by 16 bytes in memory, and is now equal to 80 bytes. Nonetheless, these two approaches compensate this overhead with the reduction of nodes and reduce the size of the BST by a factor greater than 3.

Figure 9 presents a comparison of the time spent in epochs when running MiniVite from 32 processes on 2 nodes to 256 processes on 16 nodes. Each point represents the average over 50

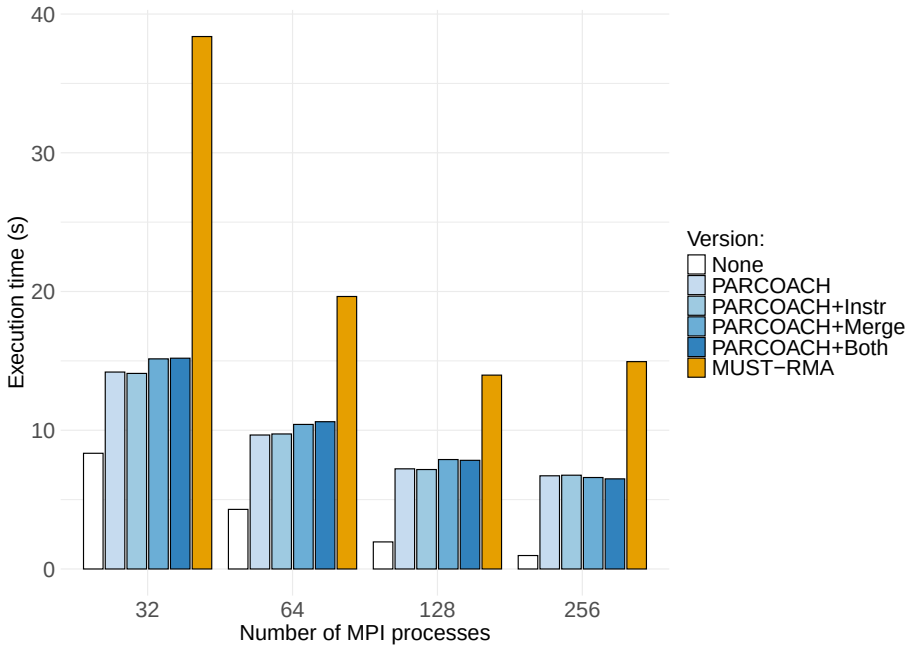


Fig. 9: Execution time of MiniVite with a problem size of 1,280,000, depending on the number of processes and the approach used.

runs. When comparing *PARCOACH* to the approaches presented in this paper, *PARCOACH-Instr* slightly reduces the runtime overhead. This is due to the reduction of instrumentations which implies that fewer instructions are analyzed by our approach. This speedup is small for the same reason as why the number of nodes is not reduced by the approach. *PARCOACH-Merge* requires more operations when inserting a new node which has a cost. This cost is not compensated by the reduction of the size of the BST. For the same reason, *PARCOACH-Both* has more overhead than *PARCOACH* because of the cost of the merging algorithm. Nevertheless, for users that have memory usage limitations, these approaches have a good trade off between performance and memory usage. Moreover, these two approaches have better performance when running at larger scale as the number of analyzed instructions is reduced and so is the cost of the insertion algorithm.

MUST-RMA has a notable overhead compared to the PARCOACH approaches. The reason is that ThreadSanitizer instruments all the memory accesses of the application.

5 Conclusion

This paper proposes an extension of PARCOACH analyses to take advantage of the static analysis to reduce the number of memory accessses instrumentation, reduce the overhead of the analysis at runtime, and support synchronizations in the data race detection and avoid false positives. Experiments have shown that these contributions lead to a better accuracy, and a reduction of the memory usage which may be useful for users that have memory boudaries. Therefore, a reduction of the execution time can be noticed at larger scale. We leave for future work an in-depth study

on the behaviour of our contributions on applications with different memory access patterns. We plan to make an interprocedural analysis for the new instrumentation analysis since the latter does not instrument instructions within a function call which may lead to false negatives. We also plan to propose an other way to promote the use of MPI one-sided communications with a code transformation solution that finds regions in the code where one-sided communication may be beneficial, and transforms MPI two-sided communications into one-sided communications.

References

1. Diep, T.D., Fürlinger, K., Thoai, N.: 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, Association for Computing Machinery, New York, NY, USA (2018)
2. Ghosh, S., Halappanavar, M., Kalyanaraman, A., Khan, A., Gebremedhin, A.H.: Exploring mpi communication models for graph applications using graph matching as a case study. In: 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 761–770 (2019)
3. Ghosh, S., Halappanavar, M., Tumeo, A., Kalyanaraman, A., Lu, H., Chavarrià-Miranda, D., Khan, A., Gebremedhin, A.: Distributed louvain algorithm for graph community detection. In: 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 885–895 (2018)
4. Hilbrich, T., Schulz, M., de Supinski, B.R., Müller, M.S.: MUST: A Scalable Approach to Runtime Error Detection in MPI Programs. In: Tools for High Performance Computing 2009. pp. 53–66 (2010)
5. Park, M.Y., Chung, S.H.: Detecting Race Conditions in One-Sided Communication of MPI Programs. In: 2009 Eighth IEEE/ACIS International Conference on Computer and Information Science (2009)
6. Saillard, E., Sergent, M., Aitkaci, T.C., Barthou, D.: Static Local Concurrency Errors Detection in MPI-RMA Programs. In: Correctness 2022 - Sixth International Workshop on Software Correctness for HPC Applications. Dallas, United States (Nov 2022), https://hal.inria.fr/hal-03882459
7. Schwitanski, S., Jenke, J., Tomski, F., Terboven, C., Müller, M.S.: 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). pp. 27–36 (2022)
8. Serebryany, K., Potapenko, A., Iskhodzhanov, T., Vyukov, D.: Dynamic Race Detection with LLVM Compiler. In: Khurshid, S., Sen, K. (eds.) Runtime Verification. pp. 110–114 (2012)
9. Simmendinger, C.: Pgas community benchmarks cfd-proxy version 1.0.1. https://github.com/PGAS-community-benchmarks/CFD-Proxy (2014)
10. Vinayagame, R., Saillard, E., Thibault, S., Nguyen, V.M., Sergent, M.: Rethinking Data Race Detection in MPI-RMA Programs. In: Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis. SC-W '23 (2023)
11. Virouleau, P., Saillard, E., Sergent, M., Lemarinier, P.: Highlighting PARCOACH Improvements on MBI. In: Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis. SC-W 2023 (2023)