Dynamic Data Race Detection for MPI-RMA Programs

Tassadit Célia Aitkaci, Marc Sergent, Emmanuelle Saillard, Denis Barthou, Guillaume Papauré

Dynamic Data Race Detection for MPI-RMA Programs

Célia Tassadit Ait Kaci
Atos, Inria Bordeaux Sud-Ouest
Echirolles, France
tassadit.aitkaci@atos.net

Marc Sergent
Atos
Echirolles, France
marc.sergent@atos.net

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

Denis Barthou
Bordeaux Institute of Technology, U. of Bordeaux, LaBRI
Bordeaux, France
denis.barthou@inria.fr

Guillaume Papauré
Atos
Echirolles, France
guillaume.papaure@atos.net

ABSTRACT

One-sided communications is a well known distributed programming paradigm for high performance computers, as its properties allows for a greater asynchronism and computation/communication overlap than classical message passing mechanisms. In this paper, we focus on the Remote Memory Access interface of MPI (MPI-RMA), in which each process explicitly exposes an area of its local memory as accessible to other processes to provide asynchronous one-sided reads, writes and updates. While MPI-RMA is expected to greatly enhance the performance and permit efficient implementations on multiple platforms, it also comes with several challenges with respect to memory consistency. Developers must handle complex memory consistency models and complex programming semantics. This paper presents the RMA-Analyzer, a new tool that detects memory consistency errors (also known as data races) during MPI-RMA program executions. It collects relevant MPI-RMA operations and load/store accesses during execution and performs an on-the-fly analysis to stop the program in case of a consistency violation. We validate our method on a collection of codes containing errors and on two real applications. Our experiments show that the RMA-Analyzer is scalable when running on MPI one-sided applications with an overhead of 40% at best.

1 INTRODUCTION

Recent advances on Exascale computing lead to deal with the increasing computational power and stress the requirement of correct and efficient concurrent programs that can be executed in parallel on multiple cores. However, ensuring the correctness of a parallel program is challenging. Concurrent programs could wage to concurrency bugs that are difficult to localize and fix.

Parallel programs can be written by using different programming paradigms. Among them, the Message Passing Interface (MPI), and the Parallel Global Address Space (PGAS) model are largely used in HPC systems. While MPI and PGAS are often referred as two different programming paradigms, MPI allows a model for remote memory access called MPI-RMA. First introduced in the MPI-2 specification, this programming model is quite similar to PGAS, as it is also based on one-sided communications of data, and global access of partitioned memory. Unlike MPI two-sided, where the sender and the receiver explicitly call the send and receive functions, one-sided communications decouple data movement from synchronization and offer asynchronous reads, writes, and updates without involving the target process. While MPI-RMA allows efficient data movement between processes with less synchronizations, its programming is error-prone as it is the user responsibility to ensure memory consistency. It thus poses programming challenges to use as few synchronizations as possible, while preventing data race and unsafe accesses without tampering with the performance.

To the best of our knowledge, very few works exist to detect concurrency bugs in MPI-RMA and they all rely on post-mortem analyses. In this paper we present RMA-Analyzer, a framework that identifies memory consistency errors in MPI-RMA programs written in C and Fortran. Our framework uses a dynamic on-the-fly analysis and focus on MPI-3 features. The advantages of our approach are twofold. First, contrary to state-of-the-art solutions, this analysis can detect all consistency errors that can happen during the execution of an MPI-RMA program, for a given set of input data. Second, when an error occurs, the analysis can directly stop the program and warn the user about detailed information on conflicting accesses. Indeed, since a silent race condition can provoke errors later in the program, it is mandatory to detect the first race condition and immediately warn the user that an error has happened, instead of waiting until the end of the program to do so. We argue that these two properties are of tantamount importance for helping users porting large-scale code bases on MPI-RMA.

The paper is organized as follows: Section 2 presents the MPI-RMA programming model and its associated programming challenges. Section 3 explains the core concepts of the RMA-Analyzer. Section 4 presents an evaluation of the RMA-Analyzer in terms of error coverage and overhead and Section 5 puts into perspective our contribution against the state of the art. Finally, Section 6 gives conclusive remarks and discusses future works.

2 BACKGROUND

This section first presents the concepts of the PGAS and MPI-RMA models. Then it discusses the programming challenges associated

to the use of MPI-RMA, and in particular the consistency errors that motivate the contribution of this paper.

2.1 PGAS model and MPI-RMA

In PGAS programs, each process exposes a part of its local memory to other processes, as shown in Figure 1.

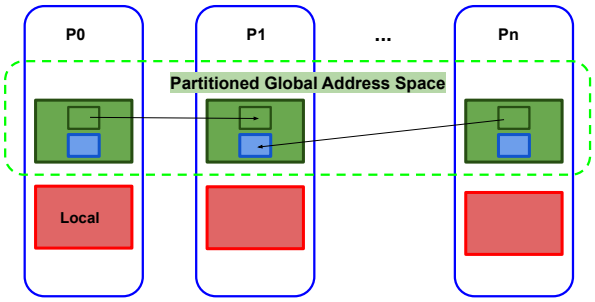


Figure 1: Overview of the Partitioned Global Address Space (PGAS) model

This way, the memory of other processes can be directly addressed from a sender, thus allowing to perform one-sided communications (e.g. Put, Get). This communication model is known to significantly improve the asynchronism and the overlap of communications with computations, which is why it is expected to gain focus in the next years for the Exascale era and beyond [**?**].

MPI-RMA is not as popular as PGAS models due to its programming complexity. The main difference between PGAS implementations, such as language-based Co-Array Fortran [**?**] or UPC [**?**] or runtime-based GasNet [**?**] or ARMCI [**?**], and MPI-RMA is the synchronization model. Indeed, one-sided communications on an MPI memory region exposed to other processes (called a *window*) can only be performed inside a so-called MPI *epoch*. An epoch enforces a type of synchronization between communications.

MPI-RMA offers two kind of synchronization modes for its epochs: *Active Target* synchronization and *Passive Target* synchronization. The main difference between them resides in whether the targets of one-sided communications are involved in the synchronization or not. In the *Active Target* mode, all processes that are involved in the epoch must synchronize. This synchronization can be either done with *Fence* or *PSCW* (Post-Start-Complete-Wait). An example of both synchronizations is presented in Figure 2. All processes can call one-sided communications between two fence synchronizations. With PSCW synchronization, two processes must explicitly synchronize. In the *Passive Target* mode, the target process does not participate in the synchronization. MPI offers two models of Passive Target synchronization: the *Lock-Unlock* model on a specific target and the global *Lock_all-Unlock_all* model as presented in Figure 3. This feature may lead to reduced safety (i.e. conflicting accesses could occur) since each process can access to all other windows at any time without any MPI call to the target process to ensure synchronization. To tackle this issue, the standard proposes a set of routines called `MPI_Win_flush*` to synchronize communications inside a Passive Target epoch. The

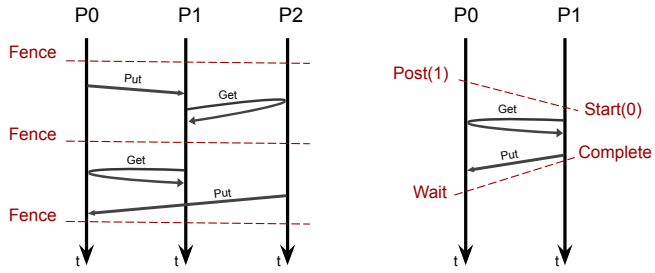


Figure 2: Examples of active target modes where synchronizations are made through fence functions (on the left) or with Post-Start-Complete-Wait (on the right).

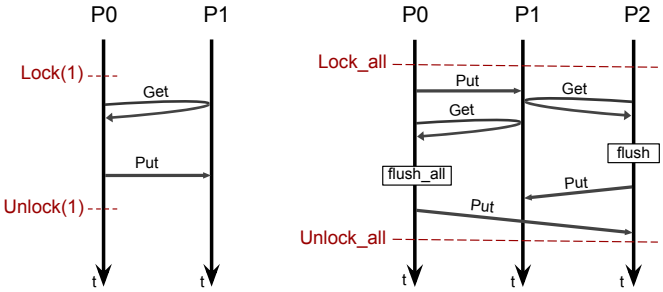


Figure 3: Examples of passive target modes where synchronizations are made through Lock-Unlock functions targeting a specific process (on the left) or Lock_all-Unlock_all (on the right).

`MPI_Win_flush_local/_all` routines ensure that outgoing communications of the calling process are completed at origin side only, while `MPI_Win_flush/_all` ones ensures that outgoing communications are completed at both origin and target side. However, only the calling process knows that the communications has completed at target side, not the target itself. Such semantics require application developers to implement their own synchronizations on top of MPI-RMA for the target side, which can greatly hinder the performance of the application.

In this paper, we choose to focus on the Lock_all-Unlock_all model of the Passive Target synchronization mode. Indeed, if a Passive Target epoch is started at the beginning of the program (i.e. just after the MPI window declaration) and stopped at the end of the program (i.e. just before freeing the MPI window), as shown in the right part of Figure 3, such MPI-RMA program behaves closely to a PGAS program, which is our target in this paper. However, the introduction of the Passive Target mode in MPI-3 brings new programming challenges, especially in the context of memory consistency, that we discuss in the next section.

2.2 MPI-RMA Programming Challenges

The MPI-RMA one-sided communication model exhibits several properties that are mandatory to achieve efficient overlap of communications with computations.

The first property is *completion*. To allow overlap of communications with computations, one-sided communications are asynchronous by nature, i.e. there is no need of progress for these communications to complete. This also means that, when initiated, the programmer has no way to know when the communication has been completed, before the end of the current epoch. The second property is *ordering*. Inside a passive target epoch, the communication ordering is not known. This can lead to erroneous programs, and is often fixed by over-synchronizing the program, causing a severe performance loss due to lack of overlapping possibilities. Finally, the third property is *atomicity*. Except for `Accumulate` routines ensuring the atomicity of accesses, regular MPI-RMA one-sided communications (e.g. `MPI_Put`, `MPI_Get`) are not atomic. This means that concurrent accesses to the same memory location result in an undefined behavior. The MPI standard does not categorize this case as an error, but specifies that this is implementation dependent. In this paper, we report such scenarios as errors.

While these three properties – completion, ordering and atomicity – are the root of the performance of MPI-RMA, the associated nondeterminism makes it difficult to implement such programs without errors. These errors can, however, be classified following a memory consistency error model. As explained by Hoefler et al. in [**?**], a correct MPI-RMA program is a program where each conflicting access is synchronized with a process synchronization, called *happens-before synchronization*, and a memory synchronization, which is called *consistency order*. Memory consistency errors were introduced by Chen et al. in [**?**]. A memory consistency error exists in a program if two conditions are satisfied. First, two concurrent operations (e.g. MPI one-sided communications or load/store operations) access to the same memory location. Second, at least one of the operations is a write access, i.e. change the content of the memory location. The outcome thus depends of the execution order. To generalize, two events A and B are said to be *access-concurrent*, or not *cohb*, when they are not ordered by a happens-before consistency order. In this case, a data race error can occur at runtime. Happens-before relation between two events A and B can be the program order within one process, and the synchronization order between different processes. Consistency order between two events A and B thus defines a partial order of the memory actions, and guarantees that the memory effect of A is visible before B. This order is necessary because some synchronizations like the `Flush` order memory accesses without synchronizing processes. The asynchronous nature of operations in MPI-RMA leads to access overlapping buffers if the consistency order is not respected.

Two kinds of memory consistency errors can happen in an MPI-RMA program. The first kind can occur inside a single process, while the other kind happens between processes. Fig. 4a describes a typical example of the first kind of error. Here the `MPI_Put` in process `P0` has no consistency order relation with the local `Store` operation. Both operations can thus potentially access the variable `buf` at the same time. This means that memory consistency errors exist between MPI-RMA operations and local memory operations in a single process.

Figures 4b and 4c illustrate errors occurring between different processes. In Figure 4b, an `MPI_Put` on process 0 and an `MPI_Get`

P0 (Origin)	P1 (Target)
	Window location X
Win_lock_all	Win_lock_all
Put(buf, 1, X)	
buf = ..	
Win_unlock_all	Win_unlock_all

(a) Consistency error in a process

P0 (Origin)	P1 (Target)
	Window location X
Win_fence	Win_fence
Put(_, 1, X)	**Get(X, 0, _)**
Win_fence	Win_fence

(b) Consistency error between two processes

P0 (Origin)	P1 (Target)	P2 (Origin)
	Window location X	
Win_lock_all	Win_lock_all	Win_lock_all
Put(_, 1, X)		**Put(_, 1, X)**
Win_unlock_all	Win_unlock_all	Win_unlock_all

(c) Consistency error between three processes

Figure 4: Examples of memory concurrency errors using passive and active target modes. Bold statements are conflicting memory accesses.

on process 1 both access the same window buffer X, located on process 1. The two operations do not have happens-before consistency order relation between them, since they could happen in any order. This means that memory consistency errors also exist when interleaving MPI-RMA operations between processes. Finally, in Fig. 4c, processes 0 and 2 both perform an `MPI_Put` at the same window location on process 1. For the same reason as before, a consistency issue also exists in this case, where different processes access a same memory location concurrently on the same peer.

3 DYNAMIC DATA RACE DETECTION IN MPI-RMA PROGRAMS

This section presents our method to detect all erroneous situations described in the previous section.

3.1 RMA Operations Compatibility

Since MPI-3, all RMA operations are allowed inside an epoch but some situations can lead to an undefined behavior. RMA operations compatibility is presented in Table 1. In this table, all operations are supposed to read or write on the same memory location. The cross means that consistency order between the two operations is not guaranteed. This table is different from the one in [**?**] as we separate operations from/to the origin and target processes. Indeed, RMA operations can be considered as READ or WRITE operations depending on which process performs them. As an example, a `Put` operation is a READ for the origin process and a WRITE for the target process.

As the completion of RMAs can occur any time during an epoch, and is not dependent on the time the RMA call is issued, if two accesses are performed to the same address during the same epoch, and one of them is a write, then there is a coherency issue. Several scenarios involving different processes can account for this situation: Figure 5 presents conflicting accesses at the (a) origin and (b)

		ORIGIN		TARGET		LOAD	STORE
		GET	PUT	GET	PUT		
O	GET	x	x	x	x	x	x
	PUT	x	✓	✓	x	✓	x
T	GET	x	✓	✓	x	✓	x
	PUT	x	x	x	x	x	x
LOAD		x	✓	✓	x	-	-
STORE		x	x	x	x	-	-

Table 1: Compatibility of RMA operations and local load-/store accesses on the same address space. O=ORIGIN, T=TARGET, ✓= overlapping is permitted, x=undefined behavior, overlapping is not permitted.

target of RMA operations. Conflicts can also arise if one memory location is used as origin (i.e., as local buffer in Get/Put) and as target window. In Figure 5a, one process can issue a distant `GET` to write a local variable in its private address space and use this variable with a load. As both accesses are unordered, this is an issue. The same situation arises when the access address is in a window: a local store for instance and a `PUT` to a distance variable, reading this same variable is also a coherency issue. Figure 5b illustrates different cases involving three processes. For instance, a distant `PUT` by one process and a distant `GET` by another process, to the same address in the same window leads to non coherent result (red edges in the figure). In all these cases, the process owning the address causing the coherency issue should raise an error and interrupt the program execution.

3.2 Data Race Detection Algorithm

To detect errors in Table 1, the runtime has to maintain a precise state of the distributed memory. We propose an RMA-Analyzer that keeps track, for each process, of all the accesses performed to memory addresses it owns and shares with other processes through RMA. Accesses can be either local loads and stores, or MPI-RMA operations, from the process itself or from remote processes.

In order to reduce the cost in term of time and space of such bookkeeping, memory regions modified are stored as union of disjoint intervals in a binary search tree (BST). Each node of the BST contains a memory address interval (*itv*), the access type (*access*) and possibly empty left (*Left*) and right (*Right*) subtrees containing intervals lower and higher resp. to the parent interval. All BST modifications are protected by a lock.

For each window created, each process creates a BST associated to the addresses it owns in this window. In order to be notified by distant processes of accesses to the addresses it owns in a window, a new thread is created and keeps calling `MPI_Recv` from any source to receive access notifications from other processes. A BST is also created for variables that are not in any window, at `MPI_Init`. For each beginning of epoch, associated to the window is emptied. When a `GET` is executed, the BST of the origin process is updated with the local variable written, either the BST of a window or the BST of local variables. The distant variable has to be added to the BST of the target process as a read. The `GET` function is instrumented to send a notification (with an `MPI_Send`) of the address read to the target process, and the process increases the number of notification sent to the target. Likewise, a `PUT` sends a notification for a write to the target process and adds a read to the local BST. In both cases the target process receives the notifications and all updates to the BST are performed according to Algorithm 1. It checks if the new memory access will lead to a concurrency error or not. If the memory access is safe, the memory interval is inserted in the BST (line 5 in the algorithm). If the memory access overlaps a memory interval stored in the BST and one of them is a write, the program is stopped and an error message is returned to the developer (line 3 in the algorithm). The same applies for `PUT` accesses. When an epoch terminates, each process counts the number of notification the other process have sent to it (with an `MPI_Reduce`) and terminates the receiver thread when all messages have been received. This behavior is made possible by the fact that, as explained in Section 2.1, we focus on MPI-RMA programs that use the epoch synchronization calls on all the processes of the window, i.e. as a collective call. Generalizing this approach by exchanging this number of notifications through `Send/Recv` calls instead would be feasible, but is out of the scope of this paper. Finally, all loads and stores are also instrumented during an epoch and accesses are registered in the BST for local variables or the BST associated to a window, depending of the range of addresses.

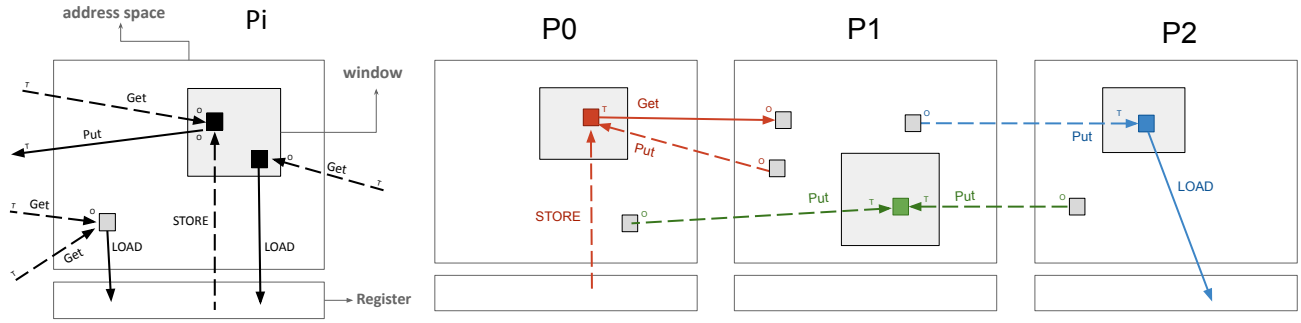
The intervals of the BST do not approximate regions: all addresses in the interval have been accessed with the access type registered. There is no over-approximation of the accessed regions and therefore no false positive.

Algorithm 1 Data Race Detection

Require: Binary search tree T, Memory interval I, Access type A
▷ A is READ, WRITE, local READ or local WRITE
Ensure: Updated T. An error message is issued in case of a data race

```
1:  procedure BSTupdate(T, I, A)
2:      if I ∩ T.itv ≠ ∅ then
3:          if ACCESS(A, T.access) == ERROR then
4:              Raise an error and stop the program
5:          else
6:              T ← splitInterval(T, I ∩ T.itv, A)
7:      I ← I − T.itv
8:      if isLeaf(T) then
9:          T ← newNode(I,A)
10:     if I ∪ T.itv is an interval and A == T.access then
11:         T.itv ← I ∪ T.itv
12:         mergeNeighboringIntervals(T)
13:     else
14:         if I < T.itv then
15:             BSTupdate(T.Left, I, A)
16:         else
17:             BSTupdate(T.Right, I, A)
```

Table 2 defines the types of access to the memory (`ACCESS` function on line 2 in the algorithm). Local access types correspond to coherent accesses within the process. No coherency issues can arise with only local accesses. A local store access to an address previously locally read modifies its access type without error. To



(a) Concurrency errors inside a process.

(b) Concurrency errors across processes.

Figure 5: Example of memory consistency errors. Dashed edges represent WRITE operations while plain edges represent READ operations on the colored boxes. *O* and *T* respectively indicate the origin and target processes.

compare two statements for memory accesses, the first access is to be read by column, and the second one by row. For example, in the Figure 4a, the first access at origin side is a PUT, thus a read in column. The second access is a STORE in row. This combinaison leads to an error. If the two statements were inverted, a STORE would have been associated to a `local write` in column and the PUT at origin in row, which would have resulted in a legal `read` access.

Line 6 splits the interval if the access A is different from the interval access. If accesses are the same, the `splitInterval` does nothing. Lines 11-12 correspond to the case where the new interval is disjoint and next to the interval T. Both intervals are then fused, and possibly with the rightmost interval of the left subtree or the leftmost interval of the right subtree of the modified node. This is what the function `mergeNeighboringIntervals` does.

ACCESS		**local read**	**local write**	**read**	**write**
O	**GET**	write	write	ERROR	ERROR
	PUT	read	read	read	ERROR
T	**GET**	read	ERROR	read	ERROR
	PUT	ERROR	ERROR	ERROR	ERROR
LOAD		local read	local write	read	ERROR
STORE		local write	local write	ERROR	ERROR

Table 2: Transition table for access types. Given an address with an access type (first row) and a new operation to this address (first column), the table defines the new access type after the operation. We assume there is no data race within a (multithreaded) process.

Correction of the algorithm. We now prove that Algorithm 1 detects all memory consistency errors, and only memory consistency errors when there is no aliasing between co-existing windows.

All memory consistency errors are detected by the RMA-Analyzer: An error happens when two accesses are performed to the same address of the same process, one of them is a write and they are unordered. Let us assume first that this address is inside a window, and only belongs to one window (aliasing between windows is not considered). It is owned by one process and this process has created a BST at window creation. One of the accesses is either a PUT or GET, distant or local, and the other access is either a PUT, GET, load or store. Loads and stores are assumed to be coherent accesses, even if multithreaded accesses are performed. The RMA calls a `MPI_Send` to the process owning the address and since all notifications are counted before closing an epoch, this notification reaches the target process and updates the BST of this window (unique). A load or store access updates the same BST of the window accordingly and since the BST is protected with a lock, modifications are serialized and the Algorithm 1 will raise an error at the second modification. Since the BST is emptied only at the beginning of an epoch (epoch creation is a collective) the BST will find that two accesses are performed to the same address with one of them a write. If the address is not in a window, the BST in charge of local addresses is used. The local address of the GET or PUT is accessed and as these functions insert their accesses to the local BST, the coherency issue is detected.

The only errors detected by the RMA-Analyzer are memory consistency errors: In order to detect the error, the two accesses have to appear in the same epoch (BST are reset between epochs), and in the same window (there is only one BST per window) or in the same private memory (local variables of a process). According to Table 2, the error can only be raised when two accesses are performed to the same address with one of them distant and one of them a write.

3.3 The RMA-Analyzer Framework

This section presents how we implemented the concepts previously explained in our RMA-Analyzer. An overview of the whole RMA-Analyzer framework is given in Figure 6.

3.3.1 Collecting Memory Accesses. The first step of our analysis collects all memory accesses of a program. To do so, we use two methods, depending on which memory accesses are recorded.

To collect MPI-RMA routines information needed by our data race analysis, we use the PMPI (Profiling for MPI) interface. From the `MPI_Win_create` call, we get the size and the base pointer of the memory region exposed to other processes, and we start tracking it. From the `MPI_Put` and `MPI_Get` calls, we get the size and the offset of the remote access, the pointer of the local access, and their respective access types (read or write). We also instrument the

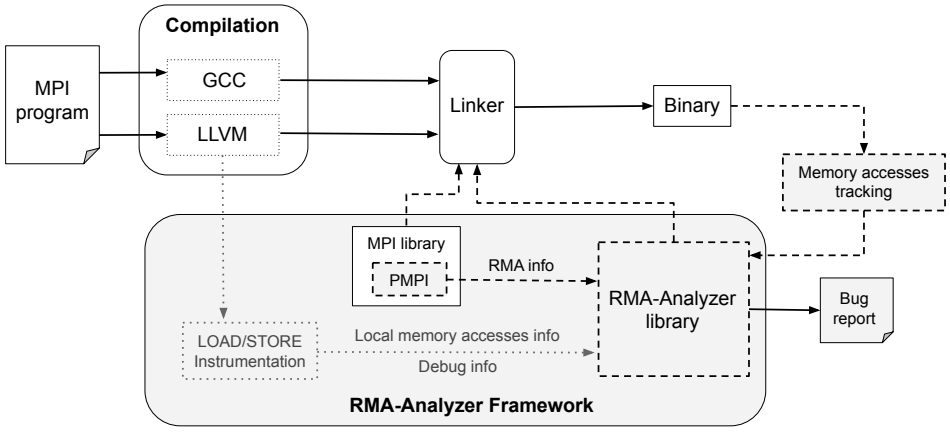


Figure 6: Overview of the RMA-Analyzer Framework.

beginning and the end of MPI epochs (i.e. `MPI_Win_lock_all` and `MPI_Win_unlock_all` in our case) to trigger and stop the recording of memory events respectively, and purge them at each end of epochs. Finally, we instrument the `MPI_Win_free` call to assess when to stop tracking the memory accesses on the associated MPI window. These pieces of information are then registered by the core of the RMA-Analyzer. While this solution is quite simple to implement and use in practice – it only needs to be preloaded at runtime through the `LD_PRELOAD` environment variable – it does not consider local memory accesses (i.e., not from a MPI routine) and cannot detect all the errors listed in Table 1.

In addition to MPI-RMA routines information, we have developed an LLVM [**?**] pass to instrument relevant `load` and `store` instructions. This enables us to detect all the errors listed in Table 1. Moreover, we implemented in this solution a compatibility with C and Fortran programs, which makes it desirable for analyzing production-quality codes that are often written in Fortran. However, it requires the user to rebuild its code to execute the LLVM pass, which can hinder its applicability for large-scale code bases.

A developer may use the GCC compiler instead of LLVM but will be limited to the detection of memory consistency errors between RMA operations. Our LLVM pass needs to be adapted to instrument `load` / `store` instructions in GCC.

3.3.2 RMA-Analyzer Core. The core of our RMA-Analyzer takes as input the memory accesses gathered by the instrumentations presented previously, and implements the data race detection algorithm presented in Section 3.2. To do so, we implemented the memory interval *itv* as a structure containing the lower and upper bounds of the memory region, and its access type. Then, we implemented a BST of memory intervals, where we aggregate both local accesses (i.e. `load`, `store` and local data accesses due to `MPI_Put` and `MPI_Get`) and remote accesses due to remote MPI-RMA calls for each MPI window. This allows the tool to compare local accesses with remote accesses, enabling it to properly cover all error cases shown in Table 1.

Local access registration is quite simple. For `load` and `store` accesses, the routine that implements the data race detection is simply called for each MPI window that is tracked by the RMA-Analyzer at the time of the access. For MPI-RMA communications, the data race detection is simply called on the local access made by the communication. Remote access registration, however, is more complex. For each MPI window, we implement a routine handled by a dedicated POSIX thread that is spawned at window creation. It is tasked to poll for all incoming communications on a specific tag range dedicated to this window by the RMA-Analyzer, and within this tag range on a specific tag identifying the MPI epoch that is currently active. This tag-based implementation is made possible by the semantics of the MPI-RMA window creation and synchronization routines, that must be performed by all the involved MPI processes for the program to be correct. This weak synchronicity allows us to implement a tag-based communication recognition system to identify all the control messages pertaining to a specific MPI epoch. Then, each time an MPI-RMA call is performed by the application program, the RMA-Analyzer adds a two-sided control message beside it (e.g. `MPI_Send`, `MPI_Irecv`) to send to the target the memory interval structure related to this memory access, with the appropriate MPI tag. Finally, when a control message is received, the thread that matches it calls the routine that implements the data race detection algorithm on its associated MPI window.

3.3.3 Implementation Concerns. While this implementation successfully detects all memory consistency errors, we need to pay attention to the overhead of such analysis on the execution time of the application. The first issue we tackle here is about instrumenting all `load` and `store` calls, that can become very costly at scale. We implement a twofold solution in the RMA-Analyzer to filter the registering of these calls. The tool only registers memory accesses performed when an MPI epoch is opened on the tracked MPI window, i.e. only when MPI-RMA calls are legal on this window. The second issue is about the active polling of communications, made by the threads dedicated to receive incoming control messages from the RMA-Analyzer. While it greatly improves the reactivity of our

tool, it can also severely hinders the performance of the application. In our implementation, we yield the thread every 100 calls to `MPI_Test` on the request associated to the `MPI_Irecv` call to release the thread as soon as possible and reduce the pressure on computational resources. It is also possible to reserve additional CPU resources so that the RMA-Analyzer threads can work on it without disturbing the application threads. Finally, it is noteworthy that while the Passive Target mode with MPI_Lock/Unlock_all is our focus in this paper, the RMA-Analyzer also supports the Fence model of the Active Target synchronization mode to ease the development of other MPI-RMA programs.

4 EXPERIMENTAL RESULTS

In this section, we first present an experimental validation of our RMA-Analyzer tool that shows that it correctly finds all the errors depicted in Table 1, and raises an error as soon as possible. Then we present an overhead study of our tool on two applications. We argue that our tool has a reasonable overhead on computations when spare cores can be used by the threads of the RMA-Analyzer.

4.1 Evaluation Methodology

Our experiments were performed on the Pise cluster that belongs to the Atos R&D department, located at Echirolles, France. Each node is composed of two AMD EPYC 7402 @2.8GHz 24-core processors with 128 GB of RAM. The nodes we used are linked with InfiniBand Mellanox 200 GB/sec (4X HDR) network cards. For the software configuration, we use a RHEL8.1 environment. Our software stack is built with LLVM 9.0.0. We use the OpenMPI implementation of MPI [**?**], built in its 4.0.5 version. The OpenMPI components are setup as follows:

- `OMPI_MCA_pml=ucx`
- `OMPI_MCA_osc=ucx`
- `OMPI_MCA_btl=^vader,openib,uct`

4.2 Validation

To highlight the functionality of the RMA-Analyzer, we created a micro-benchmark suite containing programs with correct and incorrect uses of MPI one-sided operations. This suite covers all error cases depicted in Table 1.

```
$ mpirun -np 3 ./rr_put_put
[RMA-ANALYZER Process 1] Error when inserting memory
access of type RMA_WRITE from file
remote_remote/rr_put_put.c at line 35 with already
inserted access of type RMA_WRITE from file
remote_remote/rr_put_put.c at line 35.
The program will be exiting now with MPI_Abort.
```

Figure 7: Error output returned by the RMA-Analyzer tool when runned on the code of Figure 4c.

An example of error output returned by the RMA-Analyzer when run on the code of Figure 4c is shown on Figure 7. To help the user identifying the cause of the issue in its code, the file name and file lines of the accesses causing the error, the type of these accesses, and the MPI process on which the conflict has been detected are displayed to the user before aborting the program.

4.3 Performance Evaluation

To evaluate the overhead of the RMA-Analyzer, we used two applications. The first one, CFD-Proxy [**?**], is a proxy-application for computational fluid dynamics. It implements and evaluates the overlap efficiency for halo exchanges in unstructured meshes that requires indirect read/write access via the edges of the mesh to the actual mesh data. The actual CFD kernel implements and evaluates the overlap efficiency of the two one-sided Active Target models and the two-sided model of MPI, and a GASPI version of the kernel. For our overhead evaluation, as we focus on this paper on the Passive Target mode of MPI-RMA, we retrieved the Passive Target flavors implemented by Sergent et al. in [**?**] to test with the RMA-Analyzer. This application is of interest for overhead evaluation for MPI-RMA, as it is implemented in a full `MPI_THREAD_MULTIPLE` + OpenMP model, with all OpenMP threads performing communications in parallel. This means that any overhead introduced will strongly impact the performance, which makes it a perfect candidate to evaluate our RMA-Analyzer tool in the context of strongly optimized MPI-RMA applications.

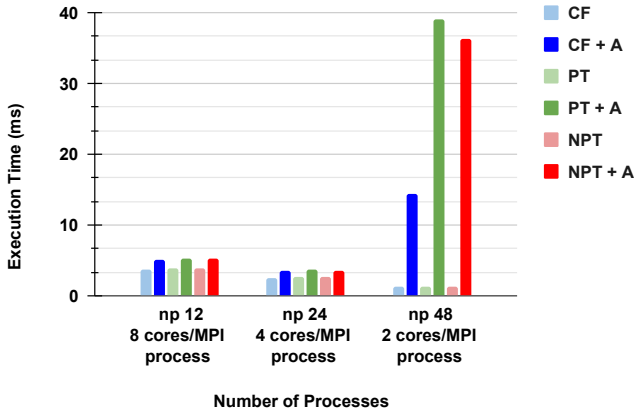


Figure 8: Runtime overhead of the RMA-Analyzer on CFD-Proxy *passive target* with tree approaches. CF = Comm Free, PT = Passive Target, NPT = Notified Passive Target. "+ A" means execution time with the RMA-Analyzer

To do so, we present in Figure 8 a comparison of CFD-Proxy runs with and without the RMA-Analyzer, on 2 nodes of our Pise cluster. For this first application, we only use the PMPI instrumentation of our RMA-Analyzer, i.e. we only instrument MPI-RMA calls. We compare three distributions between MPI processes and OpenMP threads on this configuration, from 12 MPI processes and 8 OpenMP threads to 48 MPI processes and 2 OpenMP threads. We compare three flavors of CFD-Proxy in these experiments. The Comm Free (**CF**) flavor corresponds to a run of CFD-Proxy application where all the communications are assumed to be instantaneous and with negligible overhead. When comparing the different communication schemes, the CF accords to the practical maximum attainable performance. This provides the best execution time to compare with both implementations, as it takes into account the increasing number of MPI processes, which increase the computations due

Benchmark / Application	Language	BST memory size	User window memory size	# nodes in BST	Max depth of BST
Validation test (Figure 4c)	C	0.08	0.39	2	2
CFD-Proxy	C	60	479	1500	61
NEMO	Fortran	5700	6490	142183	64893

Table 3: RMA-Analyzer statistics on BST for each application. Memory sizes are in KB.

the halo exchanges. The two other flavors stands for Passive Target (**PT**) and Notified Passive Target (**NPT**). The PT flavor uses the `MPI_Win_flush` call to synchronize the communications. The NPT one adds a flag at the end of each sent data buffer, so that each process can check this flag to ensure the completeness of the operation. It emulates a notification system for communications, thus its name. We observe two specific behaviors in this graph. For 12 and 24 MPI processes, the overhead incurred by the RMA-Analyzer stabilizes around 40%. On these distributions, the threads of the RMA-Analyzer – here 4 as CFD-Proxy uses 4 MPI windows in parallel – can use the hyperthreads of the cores used by the OpenMP threads to poll the communications, thus minimizing the impact on application threads. However, for 48 MPI processes, we see a severe degradation of the performance, with an execution time multiplied by 40 in the worst case. This is due to a lack of spare cores to use for the RMA-Analyzer threads (only 2 for 4 RMA-Analyzer threads), that will then compete with the application threads for the CPU resources. This also means that, if application developers can use spare cores during the design phase of their application, the cost of using the RMA-Analyzer remains reasonable. It is also noteworthy to recall that the overhead is not an issue for detecting errors, as the analysis is not dependent of the execution order: it detects all errors pertaining to a specific entry data set. We also tested the Active Target flavors of CFD-Proxy with our RMA-Analyzer, with similar results, thus not shown in this paper.

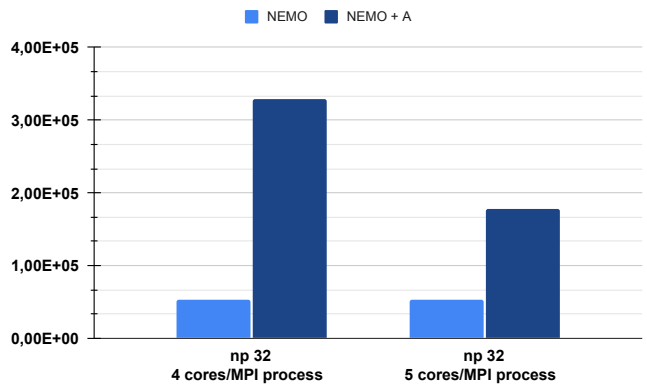


Figure 9: Runtime overhead of the RMA-Analyzer on NEMO *tra_adv* kernel. "+ A" means execution time with the RMA-Analyzer

To evaluate the overhead of the full instrumentation of the RMA-Analyzer, i.e. the PMPI instrumentation plus the LLVM pass which instruments the `load` and `store` calls, we performed similar experiments on a second application, which is a kernel extracted from the NEMO [**?**] application. NEMO – Nucleolus for European Modeling of the Ocean – is a state of the art modeling framework for oceanographic research, operational oceanography seasonal forecast and climate studies. We choose this application as we think it is a representative candidate to tackle the applicability of our contribution for production-oriented applications. The specific kernel we target in this work is the *tra_adv* kernel (Tracer Advection). We retrieved a flavor of this kernel developed internally by Atos that uses MPI-RMA communications instead of two-sided ones for performance study purposes, and run our RMA-Analyzer with it.

We present an overhead comparison of this kernel with the RMA-Analyzer in Figure 9, on 4 nodes of our Pise cluster with two distributions. In the right one, we allocate 5 cores per MPI process, so that the 4 threads allocated by the RMA-Analyzer – this kernel also uses 4 MPI windows for its computations – can use spare cores to not disturb the main application's thread. We observe that, contrary to the CFD-Proxy experiments, the overhead is already quite high, with a ~350% overhead. This is mainly due to the registering of the `load` and the `store` calls, which are performed by the main application's thread and heavily slows down the execution. For this code, the LLVM pass detects around 5.300 `load` and 2.300 `store` calls. In the second (left) distribution, where we remove one core per MPI process, we observe that the overhead grows to ~600% of the original execution time. Similarly with the CFD-Proxy application, this is explained by the RMA-Analyzer's threads that competes for CPU resources with the main application's thread, thus disturbing it heavily.

While we discussed the overhead in terms of execution time until now, we also need to take a look at the overhead in terms of memory and complexity. The Table 3 shows different statistics of our RMA-Analyzer tool for the three applications we presented in this section, from the validation test to the NEMO kernel. The third and fourth columns shows the memory fooprint of the RMA-Analyzer's BST and the user's MPI window respectively. This allows to compare the size in memory of the BST compared to the user memory region it tracks. We observe that, while the BST size seems reasonable for the CFD-Proxy application, it becomes of the same order of magnitude than the user memory region for the NEMO kernel. This is explained by the additional tracking of the `load` and `store` calls, which adds a lot of information to register in the tree. We can also observe this inflation in the fifth and sixth colums, which display the number of nodes in the BST, and its balance (i.e. the maximum depth of the longest branch of the tree). While the number of nodes was around 1500 in CFD-Proxy, it goes up to 142183 nodes with NEMO. Moreover, the balance of the BST is worse with NEMO again, with the longest branch containing almost half of the nodes of the tree. This means that the BST may need to be rotated to

ensure a good balance and reduce the complexity of inserting new nodes in it, and thus the overhead.

4.4 Discussion

The RMA-Analyzer tool could be extended in three directions. The first direction is to improve the overhead of the tool. We showed in the previous section that the overhead of the `load` and `store` calls can limit the scaling of the RMA-Analyzer. To deal with that issue, we think of filling the BST with these calls before the execution, by combining our dynamic analysis with a static analysis. This would considerably reduce the overhead of the tool due to the dynamic registering of these calls. An other solution would be to delegate the registering of these calls to the RMA-Analyzer threads. Currently, the LLVM pass instrumentation of `load` and `store` calls is performed by the main thread of the application. Delegating this work to the RMA-Analyzer threads (or an other dedicated thread) through a queue of requests to register memory access would be beneficial to reduce the overhead introduced on the main thread of the application.

The second one is the handling of in-epochs synchronization (e.g. `MPI_Win_flush`) and atomic operations (`MPI_Accumulate` and the put/get variations). Introducing the synchronizations is key to ensure the memory consistency of calls inside an MPI epoch without multiplying the epochs – thus the global synchronizations – inside the program. However, implementing the support of such synchronizations is not trivial. The semantic of the `MPI_Win_flush` and `MPI_Win_flush_local` implies that only the origin knows that communications have been completed (at origin side only or both at origin and target side). This means that we need to introduce specific control messages to warn the target that communications from a specific origin has been completed (in the case of an `MPI_Win_flush` call). Moreover, with this handling, we may also give advice to users about where to introduce those synchronizations inside a bogus program to ensure memory consistency. For atomic operations, while we believe that their access rights can be described with the taxonomy explained in this paper, the interactions between atomic and classical RMA operations can prove to be tedious to study and describe, and are thus left for future work.

The third one is the applicability of our contribution to other PGAS runtimes and implementations. Indeed, while the one-sided communication model and PGAS-based memory model of MPI-RMA is compatible with other PGAS runtimes and implementations, the main difference resides in the synchronization handling, which is based on the concept of epochs for MPI-RMA. We plan to study the necessary steps to adapt our contribution to other PGAS synchronization semantics, with a particular focus on PGAS languages to bring our contribution to end users.

5 RELATED WORK

Several approaches exist to detect errors in MPI applications including static analysis ([**?** **?**]), symbolic execution ([**?** **?**]), concolic testing ([**?**]), model checking ([**?** **?**], Aislin [**?**]), dynamic verification techniques ([**?** **?**]), MPI special libraries ([**?** **?**]) and trace-based approaches ([**?**]).

Among all the proposed tools, very few can help programmers write correct MPI-RMA programs. Marmot [**?**] and its successor MUST [**?**] use a dynamic analysis to detect several type of errors that can arise in MPI. Regarding one-sided functions, they only check if the functions parameters are valid. They don't address memory consistency errors detection. To the best of our knowledge, only three tools are able to detect memory consistency errors in MPI-RMA programs. MC-CChecker [**?**] and MC-Checker [**?**] both use a trace-based approach to detect memory consistency when using MPI RMA. MC-Checker analyses trace files to build a DAG based on the happens-before relation. This analysis does not scale well and only covers the MPI-2 standard. MC-CChecker improves MC-Checker analysis by taking full advantage of the encoded vector clock to replace the DAG. Unlike MC-Checker and MC-CChecker, our RMA-Analyzer performs an on-the-fly dynamic analysis to detect memory consistency errors between local and remote accesses and covers new functionalities introduced in MPI-3. Nasty-MPI [**?**] also relies on program profiling. It dynamically intercepts RMA calls and reschedules them into pessimistic executions. This approach forces synchronization errors rather than detecting them which is far different from our method.

On top of that, several studies have been conducted to detect data race errors in shared-memory programs. Some approaches [**?**], [**?**], [**?**] perform a static analysis, while others [**?**], [**?**], [**?**], [**?**] perform a dynamic analysis that use lock-set algorithms to detect data race errors in shared-memory programs. Some differences can be distinguished between detecting memory consistency in MPI-RMA programs and shared-memory data race detection. In MPI-RMA programs a memory consistency error can only occur in buffers that are involved in the the MPI-RMA calls. By contrast, any memory location in shared-memory programs can be a data race error location. In addition, direct load and store accesses in MPI-RMA programs are only performed by the owning process and the RMA accesses are performed among the processes in the communicator. Consequently the bug detection algorithms are different.

6 CONCLUSION AND FUTURE WORK

In this paper, we propose a novel dynamic analysis of memory consistency for PGAS-like models. We implemented it in a tool called RMA-Analyzer for MPI-RMA programs. This tool allows to detect all memory consistency errors that can happen in an MPI-RMA program and raises an error as soon as it is detected, avoiding the issue of not knowing where the first memory consistency error happened. Moreover, we show that while the overhead of our RMA-Analyzer tool is noticeable, it is possible to greatly reduce it by spreading the threads of the RMA-Analyzer on spare cores.

For future work, we envision to improve our dynamic analysis by mixing it with a static analysis. Doing so would allow us to improve the load and store filtering at compile time and detect some local errors without executing a program. We plan to integrate this mixed static-dynamic analysis in the PARCOACH framework [**?**] to benefit from the inter-procedural analysis provided by this tool to further improve our analysis. We are also looking in the direction of the MPI-RMA notified communications model, which is a novel lightweight synchronization method proposed to the MPI Forum [**?**] for MPI-RMA communications. This topic will be addressed in the context of the DEEP-SEA European project [**?**].