Static Local Concurrency Errors Detection in MPI-RMA Programs

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

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

Célia Tassadit Ait Kaci
Inria
Bordeaux, France
celia.ait-kaci-tassadit@inria.fr

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

***Abstract*—Communications are a critical part of HPC simulations, and one of the main focuses of application developers when scaling on supercomputers. While classical message passing (also called two-sided communications) is the dominant communication paradigm, one-sided communications are often praised to be efficient to overlap communications with computations, but challenging to program. Their usage is then generally abstracted through languages and memory abstractions to ease programming (e.g. PGAS). Therefore, little work has been done to help programmers use intermediate runtime layers, such as MPI-RMA, that is often reserved to expert programmers. Indeed, programming with MPI-RMA presents several challenges that require handling the asynchronous nature of one-sided communications to ensure the proper semantics of the program while ensuring its memory consistency. To help programmers detect memory errors such as race conditions as early as possible, this paper proposes a new static analysis of MPI-RMA codes that shows to the programmer the errors that can be detected at compile time. The detection is based on a novel local concurrency errors detection algorithm that tracks accesses through BFS searches on the Control Flow Graphs of a program. We show on several tests and an MPI-RMA variant of the GUPS benchmark that the static analysis allows to detect such errors on user codes. The error codes are integrated in the MPI Bugs Initiative open-source test suite.**

***Index Terms*—HPC, MPI, one-sided communication, static analysis**

I. Introduction

Communications are an essential part of HPC applications that run on supercomputers. Optimizing them and improving the overlap of communications with computations is thus of paramount importance. While two-sided communications, widely popularized by the MPI standard, is the dominant communication model, several applications use one-sided communications. Applications often use them through abstractions, such as PGAS languages, that hide the complexity of the handling of one-sided communications in terms of memory consistency and performance optimizations. However, even expert runtime programmers can face issues providing a support for such languages when using runtime systems that implement one-sided communications support. In particular, the programmer has to deal with complex synchronizations semantics that are needed to ensure the semantic and the memory consistency of the program. An example of such intermediate layer providing a support for one-sided communications is MPI Remote Memory Access (MPI-RMA), the one-sided communication interface of the MPI standard [1]. While MPI-RMA proposes a simple interface for one-sided communications (e.g. `MPI_Put/Get` and atomic operations), it also proposes several solutions to synchronize those communications through the concept of so-called MPI *epochs*. MPI epochs are delimited by specific synchronization calls, during when MPI-RMA operations can be submitted freely by the user and processed asynchronously by the MPI runtime. This behavior allows programmers to implement efficient communication/computation overlap strategies based on the asynchronous nature of MPI-RMA communications. At the end of an epoch, a synchronization of all pending communications occurs. An epoch is considered as finished only if all the MPI-RMA operations initiated or received during this epoch have been completed.

The synchronization semantics of MPI-RMA to begin and end MPI epochs can be divided in two categories. The first one, called *Active Target*, involves receiver processes in the synchronization. The synchronization models that implement it in MPI-RMA are called *Fence*, shown in Figure 1a, and *PSCW* (Post-Start-Complete-Wait), shown in Figure 1b. While some Active Target synchronization models, such as Fence, are easy to use as it is close to the well-known Barrier of MPI, which allows to easily change the communications of application codes based on Bulk Synchronous Parallel model (BSP [2]) to MPI-RMA, it suffers the same scalability issues as BSP-based codes due to the over-synchronization of the processes. Moreover, it defeats the idea of asynchronous communications and synchronizations, which is the core idea of one-sided communications to achieve performance.

The second one, called *Passive Target*, does not involve receiver processes in the synchronization. The synchronization models that implement it in MPI-RMA are called *Lock/Unlock*, shown in Figure 1c, and *Lock_all/Unlock_all*, shown in Figure 1d. In this model, in-epoch synchronizations called *Flush* are also available to perform finer-grained synchronizations. This synchronization model gives more freedom to the programmer to implement its own synchronizations at receiver side, at the cost of handling the memory consistency of the program by itself. Thus, the performance improvement comes at the cost of harder programming. Especially, race conditions are harder to detect since there is no receiver side synchronizations available.

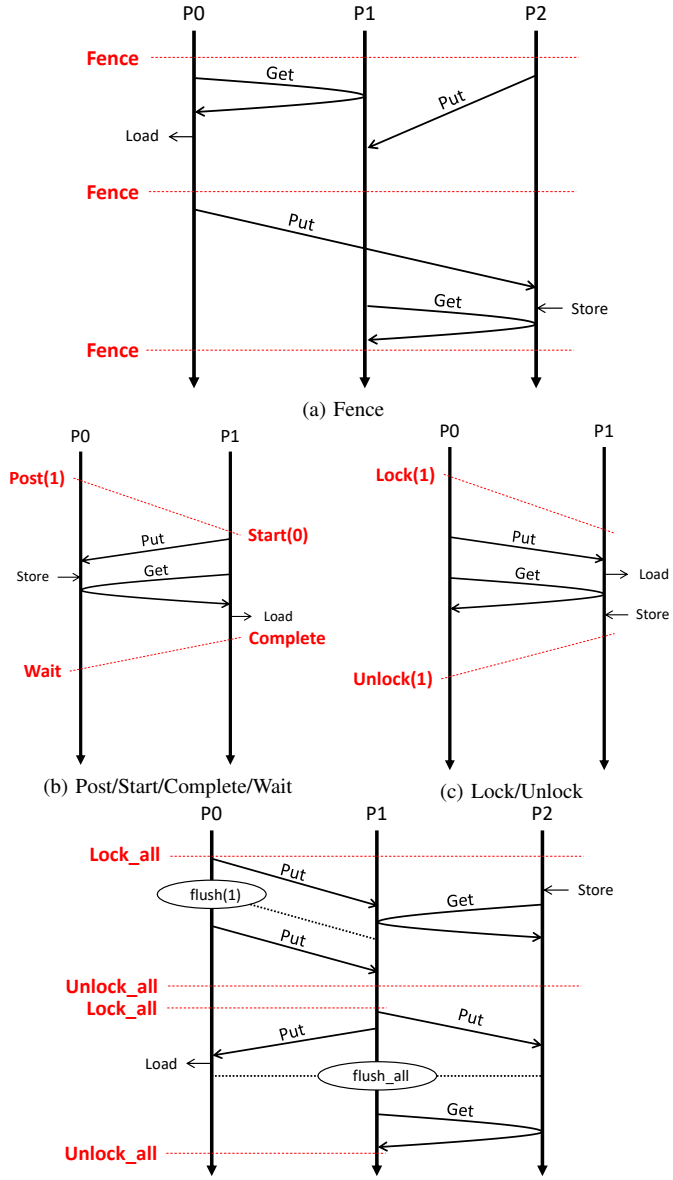

Fig. 1: Examples of use of MPI-RMA Active Target (a, b) and Passive Target (c, d) synchronization modes.

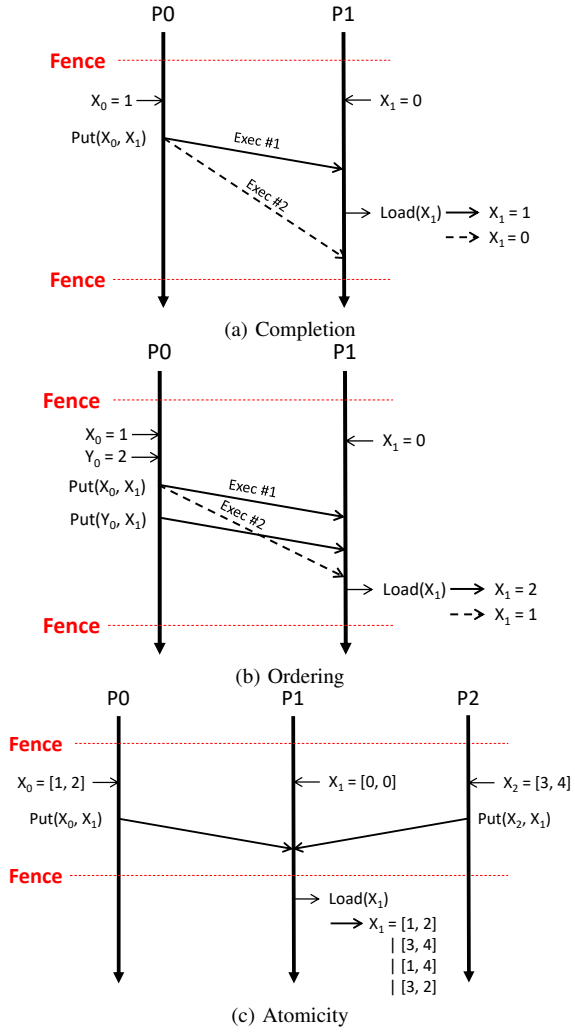

Fig. 2: MPI-RMA properties and examples of related programming challenges.

In Figure 1 we can see examples of execution sequences of MPI-RMA operations in an epoch. However, as communications can happen at any time during an epoch, these operations may overlap with other memory operations, e.g. local load/store operations performed by the program. MPI-RMA asynchronous communications are characterized by three key properties, mandatory to overlap communications with computations efficiently but which requires complex handling when programming to ensure the memory consistency of the program.

The first property relates to *completion* of communications. The completion of MPI-RMA operations is not known during an epoch. For example, in Figure 2a, the Put from process 0 can complete at any time during the MPI epoch between the two Fence calls. This means that a read on the value the `MPI_Put` call writes on can either return 1 or 0 depending on when the operation completes. A synchronization call - e.g. an end of epoch like Fence or an explicit synchronization call such as `MPI_Win_flush` - is needed to ensure the completion of MPI-RMA communications. The second property revolves around the *ordering* of communications. The ordering of MPI-RMA operations during an epoch is not defined. Communications can happen in any order, as shown in the example of Figure 2b. Depending of the execution order of the two `MPI_Put` operations, the result of the read on the value they write on can be either 2 or 1 depending of the execution order of the operations. The third property is about the *atomicity* of operations. The atomicity of MPI-RMA operations is guaranteed only at the level of an MPI datatype, except for atomic operations (e.g. `MPI_Accumulate`). An example of the type of issues that can happen is shown on Figure 2c. If a buffer composed of several elements of

a specific MPI datatype - here, two integers - are written concurrently through `MPI_Put` to the same destination, the result can be either one of them (depending of the ordering), or a combination of the two where the unit of atomicity is the MPI datatype, here an integer.

While being complex at program and use, only a few tools have tackled the issue of detecting memory consistency errors in MPI-RMA programs (referred as local and global concurrency in [3]). Most of the literature study those errors through tracing and post-mortem analysis [4], [5] To the best of our knowledge, only our previous work [6] tackles the problem of memory consistency analysis at execution time. However, the dynamic analysis suffers from the scalability issue of such approaches. In this paper, we introduce a novel static analysis that enables programmers to detect memory consistency errors directly at compile time. While the dynamic analysis is exhaustive in terms of error detection, the static analysis will be limited to local concurrency errors. However, since the analysis is performed at compile time, it scales on large code bases and can be of use for production quality codes.

The paper is organized as follows: Section II discusses the related work. Section III introduces the problem statement and describes the static analysis algorithm. Section IV presents experimental results on micro-benchmarks and an MPI-RMA variant of the Giga Updates Per Second (GUPS) benchmark [7]. Finally, conclusive remarks and future works are discussed in Section V.

II. Related Work

There is a lack of studies to detect bugs in MPI-RMA programs [3]. From the well-known projects, ITAC [8] and MUST [9] detect invalid parameters in MPI one-sided calls. In addition, these tools focus on revealing existing deadlocks in such programs. Model checking [10] is another approach that also detects latent deadlocks in MPI one-sided programs. Scalasca [11] mainly identifies potential performance bottlenecks in MPI-RMA programs. It detects inefficient wait states and highlights errors root causes by providing useful information in order to diagnose the errors. Nonetheless, among the aforementioned methods and tools, none of them addresses memory consistency errors.

To the best of our knowledge, only the following tools can detect memory consistency errors (aka data races) in MPI-RMA programs. The work in [12] resolves data races in MPI one-sided programs by leveraging mirror memory. This work misses errors as it does not consider load and store accesses. MC-CChecker [5] and MC-Checker [4] both use a post-mortem analysis to detect data races. MC-Checker builds a DAG based on the happens-before relation to analyze the trace files. This analysis does not scale well. MC-CChecker improves MC-Checker analysis by taking full advantage of the encoded vector clock to replace the DAG. Both tools do not consider the MPI-3 specifications. RMA-Analyzer [6] uses an on the fly dynamic analysis to detect data races between local and remote accesses which unlike our approach requires costly executions that may experience scalability issues. Nasty-MPI [13] is another approach that relies on program profiling in order to address so-called synchronization errors in MPI-3 one-sided applications. It dynamically intercepts RMA calls and reschedules them into pessimistic executions. This approach focuses on forcing synchronization errors rather than detecting them which is far different from our method.

While our work is related to the detection of race conditions in shared memory programs, its main difference resides in the handling of MPI-RMA specific synchronization semantics. Indeed, the data race condition problem in shared memory has been heavily addressed in the literature, but its handling is strongly linked to the synchronization model available to enforce the consistency between memory accesses. For pthread-based programs, we can cite offset-span labelling of nodes [14] or lockset-based approaches [15]–[17]. Specific contributions have been made for OpenMP to take advantage of the OpenMP synchronization semantics to enhance the analysis. While most of the literature relies on dynamic analysis [18]–[20] due to the difficulty to assess the behavior of the OpenMP runtime statically, there are a few recent work on static analysis, such as LLOV [21] and OpenRACE [22]. Since the data race detection algorithm is strongly related to the synchronization semantics that orders the memory accesses, our work differs from the state of the art by tackling the MPI-RMA specific synchronization modes to devise a detection algorithm adapted to the constraints of these semantics.

III. Static Detection of Local Concurrency Errors

A. Problem Statement

The goal of our static analysis is to detect all concurrency errors that can be detected at compile time. In this paper, we focus on all errors happening at the origin side of communications, where a program analysis coupled with an alias analysis is sufficient to find errors between overlapping accesses. Detecting errors at target side would be possible if the offset of the remote communication is completely known at compile time (in particular, if this offset is not dependent of the MPI rank), but this is out of the scope of this paper. To clearly state the errors that our static analysis can detect, we reuse the table presented in [6] and keep only the accesses at origin side, which results in Table I. In this table, we suppose that memory accesses are performed on overlapping memory regions, and we show the compatibility of local load/store operations with local buffers of MPI-RMA communications. The table should be read as follows: the first access is read in row, the second in column.

Several examples of these concurrency situations are given in Figure 3. In Code 3a, the first and second accesses are both MPI-RMA Get accesses, thus write accesses in local memory, which is an error. In code 3f, the MPI-RMA Put is the first access and the store the second one, which is also an error (the PUT in the R row and the STORE in the last column in table I is not a permitted overlap). Indeed, since the completion of the Put is not known during an epoch, as shown previously on

		READ		WRITE	
		PUT	**LOAD**	**GET**	**STORE**
R	**PUT**	✓	✓	x	x
	LOAD	✓	-	✓	-
W	**GET**	x	x	x	x
	STORE	✓	-	✓	-

TABLE I: Compatibility of RMA operations and local load/-store accesses on the same address space. ✓= overlapping is permitted, x=undefined behavior, overlapping is not permitted. R = READ, W = WRITE. The first access is read in row, the second in column.

Figure 2a, the read access on the local buffer of the operation can happen at any time between the call by the user code and an MPI-RMA synchronization call that confirms completion of the operation. However, if the accesses were reversed - i.e. the store before the Put - there would be no errors, since the store would have completed before the beginning of the Put.

```
Pi
Win_lock_all
Get(buf, target1, X)
Get(buf, target2, X)
Win_unlock_all
```

(a) Code 1

```
Pi
Win_fence
Get(buf, target, X)
... = buf
Win_fence
```

(b) Code 2

```
Pi
Win_lock_all
Put(buf, target2, X)
Get(buf, target1, X)
Win_unlock_all
```

(c) Code 3

```
Pi
Win_Fence
Get(buf, target1, X)
Put(buf, target2, X)
Win_Fence
```

(d) Code 4

```
Pi
Win_lock_all
Get(buf, target, X)
buf = ...
Win_unlock_all
```

(e) Code 5

```
Pi
Win_fence
Put(buf, target, X)
buf = ...
Win_fence
```

(f) Code 6

Fig. 3: Examples of local memory concurrency errors using passive and active target modes. Bold statements are conflicting memory accesses. X = Window location.

There are several solutions to fix these errors. Figure 4 shows correct solutions of codes 3a and 3f, either by adding an in-epoch synchronization call like `MPI_Win_flush` in the case of 4a, or by ending the epoch and starting a new one to ensure all communications have ended in the case of 4b. On both of these codes, we note that the static analysis successfully identifies the codes as correct after such changes. Codes 3b, 3d, 3c and 3e show other incorrect scenarios of the table.

B. Algorithm Description

To detect the errors presented in Table I we devised a local concurrency detection algorithm that takes place in the middle of the compilation chain. It consists in an analysis of the Control Flow Graph (CFG). The CFG is defined as a directed graph. Nodes are basic blocks that represent maximal sequence of linear code and contains a list of instructions. Edges represent the flow of control between the nodes.

```
Pi
Win_lock_all
Get(buf, target1, X)
Win_flush(target1)
Get(buf, target2, X)
Win_unlock_all
```

(a) Correction of code 1

```
Pi
Win_fence
Put(buf, target, X)
Win_fence
buf = ...
Win_fence
```

(b) Correction of code 6

Fig. 4: Correction of codes 1 (Fig. 3a) and 6 (Fig. 3f) presented in Figure 3

Based on this structure, Algorithm 1 details the steps to detect local concurrency errors in a function. First, a breadth-first search (BFS) is done on each loop of a function from the inner loops to the outer loops. Once a loop has been checked, the back edges are removed to avoid infinite loop during the BFS on an outer loop and the CFG. Finally a BFS is done on the entire CFG of a function. During the BFS, the memory accesses are spread between basic blocks (from a basic block to its successors). This is made by keeping a ValueMap containing (Value, Instruction) pairs where a *value* represents a memory access. Also, a basic block cannot be analyzed if all its predecessors have not been seen (function `mustWait` line 9 in Alg. 2). The BFS is described in Algorithm 2. The algorithm shows the BFS on a function but the principle is the same on loops. A coloring system is used during the graph traversal to identify unvisited nodes.

Algorithm 1 Local Concurrency Errors Detection

Require: CFG of function F
1: **procedure** DETECTION(Function F)
2: **for** each Loop L in F from inner to outer loop **do**
3: BFS(L) ▷ (see Alg. 2)
4: Remove back edges in L
5: BFS(F) ▷ (see Alg. 2)

Algorithm 3 gives the analysis of a basic block. The algorithm iterates over the instructions of a basic block to find memory instructions (lines 2-3). A memory instruction is either a `MPI_Put`, `MPI_Get`, a `LOAD` or a `STORE`. When a memory instruction is found, we retrieve the memory location (line 4) and check if another access on this memory location has been recorded (line 5). Function `find` line 5 returns the previous memory access on `locmem` or a memory access that alias with it. If there is no previous access, we register it in MemInBB of the current basic block but only if it is a one-sided operation (lines 6-7). We don't need to keep load/store instructions. If there is a previous access, two options arise. If the previous access or the new one is a write, we report an error (line 9-10). Otherwise, we update the last instruction accessing `locmem` if it is a Put (we don't register load instructions).

Algorithm 2 Breadth-first Search

```
Require: CFG of function F, ValueMap MemInBB (contains
    {Value, Instruction} pairs)
 1: procedure BFS(Function F)
 2:     Set each BasicBlock BB in F at WHITE
 3:     Unvisited = {}                    ▷ List of basic blocks
 4:     Add entry block of F in Unvisited
 5:     while Unvisited.size() > 0 do
 6:         header ← Unvisited.begin()
 7:         Remove header from Unvisited
 8:         if header is BLACK then continue
 9:         if mustWait(header) then
10:             Add header in Unvisited; continue
11:         UpdateMemAccesses(header)        ▷ (see Alg. 3)
12:         set header to GREY
13:         for all successors Succ of header do
14:             if Succ is WHITE then
15:                 Succ.MemInBB = header.MemInBB
16:                 Add Succ in Unvisited
17:                 set Succ to GREY
18:             else
19:                 Succ.MemInBB.insert(header.MemInBB)
20:         set header to BLACK
```

Finally, if a synchronization instruction (i.e., end of an epoch like `MPI_Win_fence`, `MPI_Win_Flush` or `MPI_Win_Flush_all`) is found, MemInBB is reset.

Algorithm 3 Analysis of a basic block - Detection of errors at origin side of communications

```
Require: ValueMap MemInBB                 ▷ (see Alg. 2)
 1: procedure UpdateMemAccesses(BasicBlock BB)
 2:     for each Instruction I in BB do
 3:         if I is a memory instruction then
 4:             locmem ← getLocalMemAccess(I)
 5:             PrevAccess ← find(BB.MemInBB, locmem)
 6:             if !PrevAccess AND I is an MPI-RMA then
 7:                 add (locmem,I) in BB.MemInBB
 8:             else
 9:                 if isWrite(PrevAccess) OR isWrite(I) then
10:                     Report an error
11:                 else
12:                     if I = MPI_Put then
13:                         PrevAccess ← I
14:         if I is a synchronization instruction then
15:             clear BB.MemInBB
```

C. Example

Figure 5 presents a CFG extracted from a benchmark computing a binary tree broadcast algorithm. This example contains a `MPI_Get` in a loop (node `if.then9`). Due to space limit, only relevant instructions are shown in the basic blocks. The algorithm first analyzes the loop {`while.cond`, `while.body`,`if.then9`, `if.end11`}. The BFS updates the memory accesses from the header of the loop (`while.cond`). `MemInBB` in the header is empty at the beginning of the BFS. The load instruction thus does not conflict with any other memory access. Nothing is registered when analyzing `while.body`. `MemInBB` is updated for `if.then9` with the `MPI_Get` instruction and `%9` which is the local memory location associated. When we encounter `while.cond` the second time, the load instruction is reported as conflicting with `MPI_Get`. `MemInBB` is updated for `while.cond` with the Get instruction. The algorithm then removes the back edge `if.end11` → `while.cond` from the loop. A new BFS is performed on the entire CFG without the back edge. The loop is analyzed again to report concurrent accesses with the beginning of the graph if any. The Get function can conflict with itself at the second iteration of the loop. An error is then reported for that. No other local concurrency is detected.

D. Discussion

Algorithm 1 is correct if all local concurrency situations are captured. We consider only Put and Get operations as Accumulate operations can be considered as a Put at the origin side. The main constraint we have is that our analysis is intra-procedural and is thus limited to the scope of a function. An inter-procedural analysis is left for future work. We then assume that a function contains all memory information needed to detect a data race. As our goal is to detect local concurrency errors, we only need to record memory accesses in an epoch. The first BFS on all loops ensures the detection of concurrency errors inside them and set the memory accesses in the headers of the loops. Then, the BFS on the entire CFG takes into account another iteration in loops and finishes because back edges are removed. All basic blocks are encountered and thus we register all memory accesses in a function. The precision of the analysis is related to the alias analysis we use to detect aliasing. The alias analysis of LLVM is conservative and false positives are possible.

Algorithm 4 only detects local concurrency errors at the origin side of the communications. As a first step towards detecting errors that can occur at the target side of communications, algorithm 4 proposes a light analysis that detects concurrent accesses on a specific target. During the graph traversal (algorithms 1 and 2), we keep a list of one-sided communications per target(`RemoteAccesses`). Each time a communication is encountered, we retrieve the target and access type (write for a Put operation and read for a Get operation) and check if the communication will conflict with a previous operation (line 6 in alg. 4). The list of one-sided operations is reset for a specific target when a synchronization is found (i.e., `MPI_Win_Flush` and `MPI_Win_Flush_local` on target or `MPI_Win_Flush_all`). The algorithm enables the detection of errors presented on Figure 6. In these examples, a process `Pi` calls two one-sided operations on the same target.

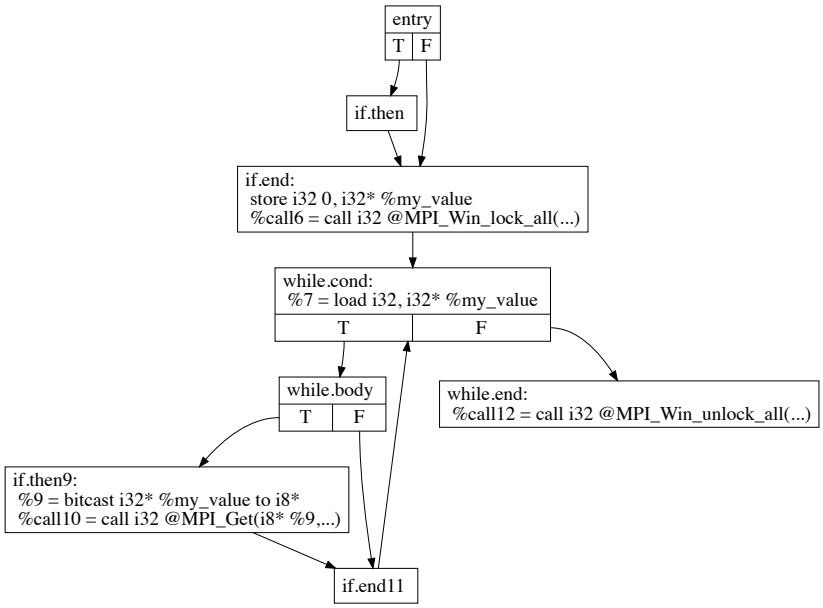

Fig. 5: CFG from a benchmark computing a binary tree broadcast algorithm.

Algorithm 4 Analysis of a basic block - Detection of errors at target side of communications

Require: RemoteAccesses: list of Instructions per MPI rank

```
1: procedure UpdateMemAccesses(BasicBlock BB)
2:     for each Instruction I in BB do
3:         if I is an MPI-RMA then
4:             target ← getTarget(I)
5:             access ← getRemoteAccess(I) ▷ write or read
6:             if isWrite(access) OR RemoteAccesses[target]
   contains a write access then
7:                 Report an error
8:             Add I to RemoteAccesses[target]
9:         if I is a synchronization instruction on target then
10:            clear RemoteAccesses[target]
```

```
Pi
Win_lock_all
Get(buf1, target, X)
Put(buf2, target, X)
Win_unlock_all
```

(a) Code 7

```
Pi
Win_Fence
Put(buf1, target, X)
OP(buf2, target, X)
Win_Fence
```

(b) Code 8 (OP=Get or Put)

Fig. 6: Examples of local memory concurrency errors at the target side of communications

IV. Experimental Results

Our analysis is implemented as a pass in the LLVM [23] framework 9.0 and uses Flang based on this version. We use the basic alias analysis of LLVM to detect aliasing on memory accesses. The pass detects errors in C, C++ and Fortran codes.

```
$ mpicc -c -g -emit-llvm ll_get_get.c
$ opt -basicaa -load analysis.so ll_get_get.bc

[STATIC ANALYSIS] LocalConcurrency detected:
conflict with MPI_Get line 38 in ll_get_get.c AND
MPI_Get line 35 in ll_get_get.c
```

Fig. 7: Output returned by the analysis on the code Figure 3a.

To highlight the functionality of our analysis, we created a microbenchmark suite containing small programs with correct and incorrect use of MPI-RMA written in C and Fortran. The suite contains 34 codes in total. Table II shows a subset of the codes, the language in which they are written, if they contain a local concurrency error and if our analysis was able to detect the error. The names of the codes correspond to the order of the operations in the code. For example, `ll_get_get` is a code containing a Get operation followed by another Get operation. The first 12 codes represent all scenarios in Table I. The codes in the second part of the table are more complicated codes, such as codes with loops, and the correct codes of Figure 4. All small codes have been integrated in the MBI project [3], a collection of correct and incorrect MPI codes. They are available on Gitlab at https://gitlab.com/MpiBugsInitiative/MpiBugsInitiative. Our static analysis was able to detect all concurrency errors and does not report any false positives on the microbenchmark suite. An example of the feedback reported by our analysis is presented Figure 7.

```
1  Window buffer upd
2 /* Create a 2D table T of size (nranks x A); */
3 /* Create a table counter of size nranks; */
4
5 for each batch {
6  for each x in batch {
7
8   /* ... */
9
10  if (target!= myrank) {
11   //Insert rans in T at the right target:
12   T[target * A + counter[targ]] = rans;
13
14   /* ... */
15
16   if (counter[target] == A) {
17    uint64 *pa = T + offset;
18    /* ... */
19    *pa = A - 1;
20    // Send the row in out buff table concerning
          target to target:
21    MPI_Put(pa, A, MPI_UINT64_T, target, my_koff +
          n_buffered, A, MPI_UINT64_T, upd);
22    // local_flush:
23    if (isFlushMode())
24     MPI_Win_flush(target, upd);
25    else if (isFlushLocalMode() || isSrcComplete())
26     MPI_Win_flush_local(target, upd);
27
28   }
29  }else{
30   /* ... */
31  }
32 }
33 /* Sending the remaining updates; */
34}
```

Fig. 8: Code snippet from an MPI-RMA version of GUPS

Program Name	Language	Local concurrency	Detect?
ll_get_get	C	yes	yes
ll_get_store	C/Fortran	yes	yes
ll_get_load	C/Fortran	yes	yes
ll_get_put	C	yes	yes
ll_put_get	C	yes	yes
ll_put_store	C/Fortran	yes	yes
ll_put_put	C	no	no
ll_put_load	C/Fortran	no	no
ll_load_get	C	no	no
ll_load_put	C	no	no
ll_store_get	C	no	no
ll_store_put	C/Fortran	no	no
ll_load_get_loop	Fortran	yes	yes
ll_get_get_ok	C	no	no
ll_put_store_ok	C	no	no

TABLE II: Results on our microbenchmark suite.

We used our analysis on an experimental code of around 3500 lines of codes, written in C++. The code is based on the Global Update RandomAccess benchmark (AKA GUPS) [24], which updates memory at random locations according to a sequence of random numbers. The code we used is an MPI-RMA version of an existing code written in UPC++ [25], obtained from the UPC++ website at upcxx.lbl.gov and was modified to (1) aggregate data to avoid overheads costs, (2) relax the look-ahead constraint and (3) to run deterministically. A snippet of the code is presented in Figure 8. The code iterates over a set of batches of random numbers. Each number is routed to its corresponding target, but aggregated. Then aggregated random numbers are put to the memory of a remote rank. The code has been simplified to show only relevant instructions and the communication scheme.

Our analysis reports a local concurrency between `MPI_Put` line 21 and the store instructions `*pa = A - 1` and `T[target * A + counter[targ]] = rans` lines 19 and 12. This error happens because of the missing else statement in the conditional line 25. No flush is encountered if the conditional is false, which leads to a concurrency error at the second iteration of the `for` loop.

V. Conclusion

Communications are of paramount importance to achieve scalable performance on supercomputers. While two-sided communications is the dominant paradigm, one-sided communications have some traction in the community, especially through PGAS languages. However, it remains difficult for programmers to use directly runtime-level libraries that implements one-sided communications due to complex and error-prone memory semantics, that prevent them from obtaining the wanted performance gain. In this paper, we presented a novel static analysis that helps programmers debugging and understanding their MPI-RMA program. This static analysis is based on a local concurrency errors detection algorithm that performs a BFS search on the CFG of the program to tag all conflicting local memory accesses (loads, stores and local buffers of MPI-RMA operations) and reports a compiling error when an issue is identified. We implemented our method as an LLVM pass and showed on small tests and on an MPI-RMA version of the GUPS benchmark that our analysis can detect local concurrency errors on such codes.

We plan to associate the static analysis presented in this paper with our dynamic analysis presented in [6]. Moreover, coupling both our static and dynamic analyses could reduce the overhead of the instrumentation of the dynamic analysis on load and store, thus making the whole error detection more scalable. Our analysis detects errors at the origin side and proposes a light detection of errors at the target side. Detecting precise errors at target side would be possible if the offset of the remote communication is completely known at compile time. This is left for future work.

There exist few applications using MPI one-sided communications today. We believe our effort will help developers in the future.

References

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

[2] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[3] M. Laurent, E. Saillard, and M. Quinson, "The MPI Bugs Initiative: a Framework for MPI Verification Tools Evaluation," in *2021 IEEE/ACM 5th International Workshop on Software Correctness for HPC Applications (Correctness)*, 2021, pp. 1–9.

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

[5] T.-D. Diep, K. Fürlinger, and N. Thoai, "MC-CChecker: A Clock-Based Approach to Detect Memory Consistency Errors in MPI One-Sided Applications," in *EuroMPI'18*, 2018.

[6] T. Aitkaci, M. Sergent, E. Saillard, D. Barthou, and G. Papaure, "Dynamic Data Race Detection for MPI-RMA Programs," in *EuroMPI'21*, 2021.

[7] "GUPS (Giga Updates Per Second)," http://icl.cs.utk.edu/projectsfiles/hpcc/RandomAccess/.

[8] "Intel Trace Analyzer and Collector," https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/trace-analyzer.html, accessed: 2021-07-15.

[9] T. Hilbrich, J. Protze, M. Schulz, B. R. de Supinski, and M. S. Müller, "MPI runtime error detection with MUST: advances in deadlock detection," *Scientific Programming*, vol. 21, no. 3-4, pp. 109–121, 2013.

[10] S. Pervez, G. Gopalakrishnan, R. M. Kirby, R. Thakur, and W. Gropp, "Formal verification of programs that use MPI one-sided communication," in *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*. Springer, 2006, pp. 30–39.

[11] M.-A. Hermanns, M. Miklosch, D. Böhme, and F. Wolf, "Understanding the formation of wait states in applications with one-sided communication," in *Proceedings of the 20th European MPI Users' Group Meeting*, 2013, pp. 73–78.

[12] M.-Y. Park and S.-H. Chung, "Detecting race conditions in one-sided communication of MPI programs," in *2009 Eighth IEEE/ACIS International Conference on Computer and Information Science*. IEEE, 2009, pp. 867–872.

[13] R. Kowalewski and K. Fürlinger, "Debugging latent synchronization errors in MPI-3 one-sided communication," in *Tools for High Performance Computing 2016*, C. Niethammer, J. Gracia, T. Hilbrich, A. Knüpfer, M. M. Resch, and W. E. Nagel, Eds. Cham: Springer International Publishing, 2017, pp. 83–96.

[14] J. Mellor-Crummey, "On-the-fly detection of data races for programs with nested fork-join parallelism," in *Supercomputing'91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*. IEEE, 1991, pp. 24–33.

[15] D. Engler and K. Ashcraft, "RacerX: Effective, static detection of race conditions and deadlocks," *ACM SIGOPS operating systems review*, vol. 37, no. 5, pp. 237–252, 2003.

[16] P. Pratikakis, J. S. Foster, and M. Hicks, "Locksmith: context-sensitive correlation analysis for race detection," *Acm Sigplan Notices*, vol. 41, no. 6, pp. 320–331, 2006.

[17] J. Mellor-Crummey, "Compile-time support for efficient data race detection in shared-memory parallel programs," *ACM SIGPLAN Notices*, vol. 28, no. 12, pp. 129–139, 1993.

[18] S. Atzeni, G. Gopalakrishnan, Z. Rakamaric, D. H. Ahn, I. Laguna, M. Schulz, G. L. Lee, J. Protze, and M. S. Mller, "ARCHER: Effectively Spotting Data Races in Large OpenMP Applications," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 53–62.

[19] S. Atzeni, G. Gopalakrishnan, Z. Rakamaric, I. Laguna, G. L. Lee, and D. H. Ahn, "SWORD: A Bounded Memory-Overhead Detector of OpenMP Data Races in Production Runs," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018, pp. 845–854.

[20] Y. Gu and J. Mellor-Crummey, "Dynamic Data Race Detection for OpenMP Programs," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018, pp. 767–778.

[21] U. Bora, S. Das, P. Kukreja, S. Joshi, R. Upadrasta, and S. Rajopadhye, "Llov: A fast static data-race checker for openmp programs," *ACM Trans. Archit. Code Optim.*, vol. 17, no. 4, dec 2020. [Online]. Available: https://doi.org/10.1145/3418597

[22] B. Swain, B. Liu, P. Liu, Y. Li, A. Crump, R. Khera, and J. Huang, "OpenRace: An Open Source Framework for Statically Detecting Data Races," in *2021 IEEE/ACM 5th International Workshop on Software Correctness for HPC Applications (Correctness)*, 2021, pp. 25–32.

[23] "The LLVM Compiler Infrastructure," http://llvm.org/.

[24] "Randomaccess, hpc challenge benchmarks," http://icl.cs.utk.edu/projectsfiles/hpcc/RandomAccess/.

[25] J. Bachan, S. B. Baden, S. Hofmeyr, M. Jacquelin, A. Kamil, D. Bonachea, P. H. Hargrove, and H. Ahmed, "UPC++: A High-Performance Communication Framework for Asynchronous Computation," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019, pp. 963–973.

APPENDIX

A. Software availability and dependencies

The static analysis is available on Github at https://anonymous.4open.science/r/MPILocalConcurrencyDetection-D375/README.md. The source code of all small codes that implements the test cases of Table I is available in the repository and is integrated in the MPI Bugs Initiative. The GUPS code used in the evaluation section is available on demand. Note that the pass will be integrated in the PARCOACH[1] tool in the future. To build the static analysis, the needed pieces of software are:

- CMake $\geq$ 3.18;
- LLVM $\geq$ 9.0.

To use the static analysis, the needed pieces of software are:

- LLVM $\geq$ 9.0;
- An MPI-3 implementation that can use the Clang compiler of the LLVM software. For example, any Open MPI with the environment variable `OMPI_CC=clang` set will work on C codes.

The static analysis can be used on C, C++ and Fortran codes. To be able to use the static analysis on Fortran codes, an LLVM-based toolchain that has a Fortran front-end compiler that can generate LLVM IR is needed, such as Classic Flang (https://github.com/flang-compiler/flang/wiki/Building-Flang). Moreover, the MPI implementation **must** be built with this toolchain to have the Fortran modules of the MPI implementation built with LLVM. Exceptions in C++ are not supported by our analysis, we use the `-fno-exceptions` flag during compilation.

B. Building and Installing

Building the LLVM pass we made is done through a simple CMake workflow that can be launched as follows:

```
mkdir -p install build
cd build/
cmake ..
make && make install
```

If the build has ended successfully, the LLVM pass will be available to use as a library called `analysis.so` in the `build/src` directory.

C. Using the static analysis on code examples

To use the static analysis on an existing code, the typical user workflow is as follows:

1) Compile the MPI-RMA application with the LLVM pass that implements our static analysis, that will analyze all local and MPI-RMA memory accesses;
2) If the static analysis has not found any error, nothing is returned to the user. If there is an error, a user-friendly message is returned to the user so that it can identify the source of the memory consistency error;
3) After fixing the MPI-RMA application, the cycle restarts until the static analysis returns without error.

To analyze a program with the static analysis, all of its files must be passed through the LLVM pass. To do so, each source file must go through two additional steps:

1) Generate the LLVM bytecode of the file by specifying `-c -emit-llvm` to the compiler;
2) Apply the LLVM pass containing our static analysis with the `opt -load` tool to check the correctness of the code.

An example of use and error message given by the static analysis is given in Figure 7.

[1]PARCOACH is available on Github at https://github.com/parcoach/parcoach