PARCOACH Extension for Static MPI Nonblocking and Persistent Communication Validation

Van Man Nguyen*†‡
van-man.nguyen.ocre@cea.fr

Emmanuelle Saillard†
emmanuelle.saillard@inria.fr

Julien Jaeger‡
julien.jaeger@cea.fr

Denis Barthou*†
denis.barthou@inria.fr

Patrick Carribault‡
patrick.carribault@cea.fr

* Bordeaux Institute of Technology,
U. of Bordeaux, LaBRI, Bordeaux, France

† Inria Bordeaux Sud-Ouest
Bordeaux, France

‡ *CEA, DAM, DIF,*
F-91297 Arpajon, France
Laboratoire en Informatique Haute
Performance pour le Calcul et la simulation, France

***Abstract*—The Message Passing Interface (MPI) is a parallel programming model used to exchange data between working units in different nodes of a supercomputer. While MPI blocking operations return when the communication is complete, nonblocking and persistent operations return before the communication is complete, enabling a developer to hide communication latency. However the usage of these latter comes with additional rules the user has to abide to. This is error prone, which makes verification tools valuable for MPI program writers. PARCOACH is a framework that detects MPI collective errors using a static/dynamic analysis. In this paper we present an extension of PARCOACH static analysis to detect misuse of MPI nonblocking and persistent communications. Our new analysis adds the detection of four new error classes related to these types of communications. Its implementation was tested on several MPI micro-benchmarks, and on some CORAL or Mantevo benchmarks on which we observed an acceptable overhead at compile-time.**

***Index Terms*—MPI, Nonblocking Communication, Correctness, Static Analysis, Persistent Communication**

I. Introduction

Since it first went out in 1994, the Message Passing Interface (MPI) is the *de facto* standard for inter-node communications in supercomputers. MPI provides several interfaces to exchange data between working units called MPI processes: point-to-point communications involving a sender and a receiver, collectives communications involving a group of MPI processes exchanging data, or RMA communications allowing to write or read directly from another MPI process memory. For an application spanning over the thousands of nodes of a supercomputer, these data communications can be very time consuming.

MPI nonblocking and persistent communications are an important part of the MPI standard. They are designed to allow hiding the communication costs with other work. All communications obey to some rules in the MPI standard. It is the responsibility of the user when inserting the corresponding procedure calls to abide to those rules. Because of their design, nonblocking and persistent operations ask for extra care when using them, and especially the handling of arguments given to them. As an example, a nonblocking operation is divided in two procedure calls: an initiation call and a completion call. Between these calls, linked together with a specific MPI structure, it is erroneous to use said structure for another operation, or to access some specific other operation arguments. This is error prone, and detecting misuse of these communications at compile time can be very beneficial for MPI program writers.

PARCOACH [1]–[3] is a framework built on top of LLVM to detect misuse of collectives in MPI programs. A first analysis studies the program at compile-time and issues warnings when a potential error is detected. Then a runtime check is performed to verify all potential errors during execution. In previous works, PARCOACH was designed to detect incorrect ordering of MPI blocking collectives calls. Then it had been adapted to handle ordering of blocking and nonblocking collective calls, but no data-flow analysis was performed to detect other nonblocking related errors.

In this paper, we propose an extension of PARCOACH to detect misuse of nonblocking and persistent operations in MPI programs, including MPI persistent collectives operations voted in to be part of the next MPI standard. Based on a new data-flow analysis, PARCOACH is now able to match nonblocking and persistent initialization calls to most of the other corresponding calls. Once the matching is done, it is then possible to detect wrong management of the operation arguments. Our new analysis is fully automatic and integrated in the tool, implemented as an LLVM pass.

Section II presents the semantics of nonblocking and persistent operation, and their potential misuse. Section III describes existing works on the verification of these operations while Section IV presents our new algorithms to realize some matching for nonblocking and persistent operation procedure calls, and augment the error detection coverage of PARCOACH for such operations. Section V shows results on several benchmarks and Section VI concludes our work.

II. Usage of MPI nonblocking and persistent operations

A. Semantics and use cases of MPI nonblocking calls

An MPI nonblocking operation is composed of two procedure calls.

The first procedure call initiates the nonblocking operation. More specifically, it hands over the argument lists to the operation, including the contents of the data buffers if any, and it attaches the operation to the given request. In most cases, MPI nonblocking initiating procedure names are of the form `MPI_I<operation>` (e.g., `MPI_Ibcast`).

The second procedure call completes and frees the operation. It returns the control of the argument list, including the contents of the data buffers. The completion call can be either a `MPI_Wait`, a `MPI_Test` or `MPI_Request_free` (only for point-to-point communications). `MPI_Wait` waits until the resources needed by the communication can be safely used while `MPI_Test` tests whether the communication has completed. Those completions also exist in three additional versions: `all`, `any`, and `some`. While the basic form only completes the asynchronous communication it is associated to, the `all` derivative can be associated to multiple nonblocking communications and will complete all of them. Similarly, the `any` and `some` derivatives can be associated to multiple initialization calls and will respectively complete any of those, or at least one of those. In all derived versions the requests are freed according to the communications that have been completed [4].

Unlike blocking calls that only return when the resources of the communications can be reused, nonblocking calls do not offer such warranties. They will return as soon as they can, leaving those resources, such as the data buffers, in a vulnerable state. Depending on the nature of the communication, any access or modification of resources might lead to nondeterministic behaviors. Those issues are a supplementary burden put on the developer, leading to longer development and debugging times. In Section II-C, we expand on some of those issues that a developer should be aware of when writing code with asynchronous communications.

B. Semantics and use cases of MPI persistent calls

An MPI persistent operation is composed of four procedure calls.

The first procedure call initializes the persistent operation. More specifically, it hands over the argument lists to the operation, and it attaches the operation to the given request. However, contrary to the nonblocking operation, this first procedure call doesn't hand over the contents of the data buffers, if any. The user remains free to change the data buffer contents until a call to a starting procedure. In most cases, MPI persistent initialization procedure names is of the form `MPI_<Operation>_init` (e.g., `MPI_Bcast_init`).

The second procedure call starts the operation. After this call, the contents of the data buffers are handed over to the operation. It should not be modified by the user, nor read in the case of an output data buffer, until the call to a completing procedure call. The starting call can be either `MPI_Start` or `MPI_Startall`. Once the starting call is done, the communication involved in the operation algorithm can take place at any time, until the end of the completion call.

The third procedure call completes the operation. It returns the control of the contents of the data buffers. The completion call for a persistent operation can be the same as for a nonblocking operation. However, once the operation is completed, it can either be restarted with a new call to a starting procedure, or it can be freed with a call to freeing procedure.

Hence, the fourth procedure call to `MPI_Request_free` frees the operation by relinquishing all the arguments associated with the operation, and marking the MPI request as free and being reusable.

As with nonblocking calls, most resources passed to the initialization calls cannot be reused until the operation is freed. This can lead to errors and race conditions. These potential issues are described in the next section.

C. Type of errors

This section gives five types of errors related to nonblocking and persistent communications: Collective mismatch, missing wait, unmatched wait, request overwriting and buffer data race.

1) Collective Mismatch: MPI nonblocking collective operations, as well as MPI persistent collective operations, follow the same restrictions as their blocking counterparts. In particular, every process in a communicator must call the same sequence of blocking collective, nonblocking collective initiation and persistent collective initiation procedures. If one process has a different sequence than the others, a deadlock can arise. The code in Listing 1 describes an example of collective mismatch. Suppose this code is executed with at least 2 MPI processes, rank 0 initiates a nonblocking broadcast and calls a reduce while the other processes first call a reduce and then initiate a nonblocking broadcast. This kind of mismatch is not permitted by the MPI standard, and this code might deadlock.

Similarly, in Listing 2 rank 0 calls a nonblocking broadcast while other processes call a blocking broadcast. This is an error as nonblocking collective operations do not match with their blocking counterpart.

Listing 1: MPI Collective Mismatch Example 1

```
MPI_Request req;
if (rank == 0) {
  MPI_Ibcast(&da, count, datatype, 0, com, &req);
  MPI_Reduce(&dsend, &drecv, 1, datatype, Op , 0, com);
} else {
  MPI_Reduce(&dsend, &drecv, 1, datatype, Op, 0, com);
  MPI_Ibcast(&da, count, datatype, 0, com, &req);
}
MPI_Wait(&req, MPI_STATUS_IGNORE);
```

Listing 2: MPI Collective Mismatch Example 2

```
MPI_Request req;
if (rank == 0) {
  MPI_Ibcast(&da, count, datatype, 0, com, &req);
} else {
  MPI_Bcast(&da, count, datatype, 0, com);
}
MPI_Wait(&req, MPI_STATUS_IGNORE);
```

2) Missing wait: Any nonblocking and persistent starting call must be associated with a completion call to ensure the communication resources can be safely reused. The code shown in Listing 3 is erroneous since `MPI_Bcast_init` initializes a persistent broadcast which is then started with `MPI_Start`, but is never completed afterwards.

Listing 3: Missing Wait

```
MPI_Bcast_init(&da, count, datatype, 0, com, info, &req);
MPI_Start(&req);
/* ... */
MPI_Request_free(&req);
```

3) Unmatched wait: According to the MPI specification, an unmatched or redundant wait is not an error: a completion call is allowed to take an empty request. However we consider that it can be useful to raise this situation and warn the developer since it can hide other issues such as an unmatched initialization call. Listing 4 shows such a case. The last `MPI_Wait` is always called on a null request.

Listing 4: Unmatched Wait

```
MPI_Request req;
if(..){
   MPI_Ibcast(&da, count, datatype, 0, com, &req);
   MPI_Wait(&req)
}
MPI_Wait(&req)
```

4) Request Overwriting: Once a request has been taken by a nonblocking or persistent communication, it should not be overwritten by another statement or used by any other nonblocking or persistent communication. Since it contains information about the former communication, its corruption can prevent the completion of the said communication. As a consequence, the code presented in Listing 5 is incorrect.

Listing 5: Request Overwriting

```
MPI_Ibcast(&da, count, datatype, 0, com, &req);
MPI_Ibcast(&e, count, datatype, 0, com, &req);
MPI_Wait(&req, MPI_STATUS_IGNORE);
```

5) Buffer Data Race: Nonblocking and persistent initialization calls return as soon as possible to their caller, and they do not ensure the safety of the resources that are needed by the message. Depending on the nature of the operation and on how those resources are being used, race conditions might happen, thus leading to a nondeterministic behavior. A data that is needed for an outbound message will be sensitive to writings. As illustrated in Listing 6 the first statement below the `MPI_Isend` operation only reads the buffer `da`. However, the following statement writes to that buffer and can cause a race condition. On the other hand, a data that is being received is akin to a writing to that memory space, making any type of access unsafe.

The same problem may arise with persistent operations if the buffer is modified between the start and the completion calls.

Listing 6: Buffer Data Race

```
MPI_Request req;
MPI_Isend(&da, 1, MPI_INT, 1, tag, com, &req);
a = b + da;
da = 3;
MPI_Wait(&req, MPI_STATUS_IGNORE);
```

III. Related Work

We roughly split existing works that verify MPI applications in three categories: Dynamic analyses, Static analyses and hybrid approaches.

a) Dynamic Analyses: Dynamic analyses are applied during the execution of a program. They are able to provide accurate results with very few false-positives as they constantly monitor the state of the program. Errors are then only reported when they are about to occur. MUST [5] performs deadlock detection by building a wait-for graph which depicts the scheduling dependencies between processes. As all dynamic tools, MUST only stands for a specific environment and can miss errors (e.g., data races [6]). This is a major drawback in a high performance environment where computations are meant to be run with many parameters and can lasts for weeks. DAMPI [7] is based on a time-out approach using Lamport-clocks to detect deadlocks. It can produce false positives and suffers from the same limitations than MUST. SimGridMC [8] is a model checker for MPI applications, it checks if a program satisfies a given property (e.g., liveness, communication determinism) by considering all possible executions. Although SimGridMC identifies data races, it can't detect high level errors like unmatched waits.

b) Static Analyses: Static tools are run at the compilation of each translation unit, or at link-time in order to perform a whole program analysis. They are also completely independent of the program inputs but can lead to many false positives. MPI-Checker [9] is based on the Clang Static Analyzer. It can perform path-sensitive checks to find erroneous matchings of nonblocking communication calls as well as missing initialization, completion calls and request overwriting. MPI-Checker does not support buffer data race nor collective mismatch detection and does not check for persistent operations misuse. Ye *et al.* [6] developed a tool that uses partial symbolic

execution to detect MPI usage anomalies. It is limited to communications on `MPI_COMM_WORLD`, doesn't detect missing wait and doesn't check collectives and persistent operations. CIVL [10] and MPI-SV [11], [12] both combine symbolic execution with model checking but unlike MPI-SV, CIVL does not support nonblocking operations.

c) Hybrid Approaches: Hybrid approaches combine a static analysis with a dynamic one. This trade-off allows the best of both approches. PARCOACH [2], [3] uses this method to find collectives mismatch. It raises warnings for potential errors with debugging information like the conditionals responsible for them. The static phase is completed by an instrumentation of potentially faulty communications that will properly terminate the program and provide useful feedback if the potential error is actually a true positive. PARCOACH is mainly focused on collectives and is not able to detect any error presented in Listings 3-6. In [1], we presented a light analysis that checks if each nonblocking initiation can be matched with a completion call. This check was done by counting the number of initiation and completion calls on each path of the program and did not consider requests. Our new analysis extends PARCOACH to detect all errors presented Section II-C. To the best of our knowledge, our static analysis is the first one that detect all errors presented and tackles MPI persistent communications.

IV. Static detection of MPI nonblocking and persistent communication misuses

This section provides an in-depth description of the methods we implemented to perform a compile-time conservative verification of MPI nonblocking and persistent operation usage. We first associate the initialization calls to their completion calls using techniques that analyze the control flow of the program. After the matching is done, a basic data-flow analysis is performed to find statements that can lead to race conditions inside each overlapping window. Collective mismatch detection is performed by PARCOACH using the algorithm presented in [1] and extended in [3]. No extensions have been made to the dynamic step of PARCOACH, which still focuses on catching mismatching collective sequences.

A. Matching of procedure calls for a nonblocking operation

Nonblocking initialization and completion procedure calls are linked to each other through the request object. For each nonblocking initiation call in the current function, we look at its request and build the list of every potential completion points.

According to the MPI specification, any nonblocking initiation call must be terminated by a completion call. In other words, if the control flow goes through an initialization call, then it must flow through a completion call at a later point in the code before exiting the program or the MPI environment. A nonblocking communication may require the insertion of multiple matching completion calls in the code, as shown in Figure 1a. In this example, assuming that all MPI calls (i.e., the `MPI_Ibcast` call and both `MPI_Wait` calls) share the same request, and that the branching condition in node B is dependent of the rank of the MPI process, then the program represented by that control flow graph (CFG) is correct. Whichever branch is taken before exiting the program, the `MPI_Ibcast` call will be completed by one of the `MPI_Wait` calls. It means that, when performing our static analysis, the initialization call needs to be matched with both completion calls.

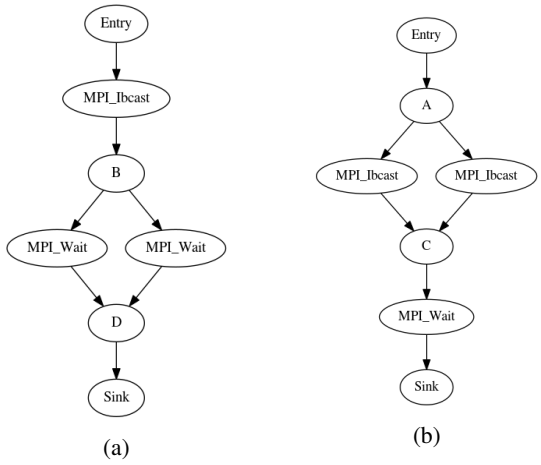

Fig. 1: CFG examples with multiple initialization or completion points for a communication

The notion of post-domination can be used to realize such static matching. A node v post-dominates a node w in a CFG if all paths from w to the exit node contains v [13]. In our situation w is the initialization call and v the completion call. When there are multiple completion calls, the notion of generalized post-domination enables us to find all sets of nodes V that post-dominate a node w. Those sets need to follow those two conditions [14]:

1) all paths from w to the end of the program must contain a node from V;
2) for each node $y \in V$, there is at least one path from w to the end of the CFG that contains y and that does not contain any other vertex in V.

In other words we need to find, for each nonblocking initiation call, the "nearest" set of completion calls that post-dominates it. The post-dominator sets are found using an adapted DJ-graph [15]. For example in Figure 1a the `MPI_Ibcast` call is post-dominated by 4 sets of cardinality 1 - itself, node B, node D, and the sink - and by one set of cardinality 2 which is the set defined by both `MPI_Wait`.

As explained in Section II-C, completion calls can intercept empty requests however we will still warn about those situations since they can help in resolving other errors. In order to catch a valid request, completion calls have to be executed only if a nonblocking communication has been initiated. In other words, the completion call needs to be dominated by an initialization call. For example, assuming that all MPI nonblocking calls share the same request object, the code represented by the CFG in Figure 1b is correct. If

the branching condition in node A depends on the rank of the MPI process, then the wait has to be matched with both initializations.

Algorithm 1 Matching MPI initiations and completion calls

Require: mpi_i: list of MPI nonblocking initiation calls in f, mpi_w: list of MPI nonblocking completion calls, dominator and post-dominator trees of f
Ensure: Each MPI nonblocking call is properly matched

```
1:  procedure MATCH_MPI_NONBLOCKING(function f)
2:      for all mpi_i ∈ f do
3:          gen_pdom ← get_genPDomSet(mpi_i)
4:          wait_set ← get_wait(mpi_i.req)      ▷ Set of waits using the same request
5:          potential_wait_sets ← ∅
6:          for all V ∈ gen_pdom do
7:              if V ⊂ wait_set then
8:                  potential_wait_sets.insert(V)
9:          if potential_wait_set ≠ ∅ then
10:             for all V ∈ potential_wait_sets do
11:                 cumul_dist = 0
12:                 for all y ∈ V do
13:                     dist = dist(mpi_i, y);
14:                     cumul_dist += dist
15:                 set_dist = cumul_dist / card(V)
16:                 if matched_dist > set_dist then
17:                     matched_dist = set_dist
18:                     matched_wait ← V
19:             check_race_cond(mpi_i, matched_wait)
20:         else
21:             Raise warning about missing wait
22:     for all mpi_w ∈ f do
23:         gen_dom ← get_genDomSet(mpi_w)
24:         mpi_i_set ← get_matched_init(mpi_w)   ▷ Set of matched initiation computed from the previous loop
25:         if mpi_i_set ∉ gen_dom then
26:             Raise warning about unmatched wait
```

The matching of MPI nonblocking initiation calls to completion calls is described in algorithm 1. As input arguments, the algorithm takes all MPI initiation and completion calls in a function. For each MPI initiation call, we compute the set of all its generalized post-dominators sets, and we get the set of all completion calls using the same request *wait_set* (lines 3 and 4). This set is determined using the def-use chains of the request object.

Since the generalized post-dominators sets can contain any statement in the function, we first prune this ensemble to keep only the sets containing nodes also present in *wait_set*, i.e., sets composed only of completion calls (lines 5 to 8). If no such set can be found, then the nonblocking initiation call cannot be matched, and we raise a warning about missing completion calls for this initiation call (lines 9 and 21). If compliant generalized post-dominators sets exist, they are stored in *potential_wait_set* (line 8).

It is then necessary to find which set is indeed composed of the actual matching completion calls. Each set of *potential_wait_sets* can properly complete the communication since they post-dominate the initiation. However, only the set of completion calls that will be visited first will be matched with the initiation. The idea is to fetch, among those sets that all post-dominate the initiation call, the one that is post-dominated by all of the remaining ones. To that end we compute the distance of each set of completion points to the initiation point, and get the set that has the shortest distance (lines 11 to 18). The distance of a set is computed by averaging the distances of each node to the initiation node. The distance of an initiation node to a wait node is the shortest path in the CFG in terms of statements.

Once we matched the initiation call to its closest completion calls, we apply the algorithm to find potential misuse of the operation arguments (line 19). This step is depicted in algorithm 2 and described in Section IV-C.

At the end of the algorithm, another pass is applied on the completion calls to check if they correctly terminate the nonblocking operation initiation they were matched with during the previous pass on the initiations calls (line 22 to 26). A completion call must be dominated by the set of initiation calls it was assigned to, otherwise a warning will be issued for an unmatched wait.

B. Matching of procedure calls for a persistent operation

Like for a nonblocking operation, all procedure calls involved in a persistent operation (i.e., initialization, starting, completion and freeing procedure calls) are linked to each other through the request object. A persistent operation should go through each of these calls in order, with the exception that after the completion call, the request can encounter either a freeing call, or a new starting call.

For the analysis, it means that a set of starting calls should post-dominates the persistent initialization call, then a set of completion calls should post-dominates each starting call. For the completion call, it should be post-dominated by a set of either freeing calls or new starting calls. Since the purpose of algorithm 1 is to find a set of MPI completions call matching a specific nonblocking initiation call, it can be easily adapted to persistent operations. A first call to algorithm 1 can be performed to match the persistent initialization call to starting calls. Then, it can be applied again to match each starting call with a completion call. Finally, it can be used to match each completion call to freeing calls, or new starting calls. If new starting calls are found, then the algorithm will be applied recursively until all persistent operations reach a freeing call.

As the initialization procedure and the freeing procedure are called only once, finding unmatched initialization and freeing is easy and can be done in their own algorithm execution. However, for starting and completion procedure calls, as multiple of them can be used for the same persistent operation, it is necessary to keep a global set of matched called over all the algorithm invocations. Unmatched starting calls

and completions calls can be found only once all the starting-completion couple procedures have been matched over the life of the tested persistent operations.

C. Detection of overwriting

As described in Section II, both nonblocking and persistent operations hand over their argument list to the operation. These arguments should not be accessed until the operation is completed or freed. More specifically, as said in Section II-C, it is forbidden to reuse a request passed to an operation for another, until the current operation is freed. Also, depending on how it is used, data buffers should not be accessed between the starting and the completion of an operation. A reception buffer of a communication should not be read inside the overlapping interval, since its state is unknown as long as the communication has not been completed. Thus the read data may not be the correct one. It is also forbidden to write, since it may squash the received data. On the other hand, while it is safe to read a send data buffer of an operation once it has been started, it is forbidden to write to it. Between the starting and the completion, there is no way to know if the writing in the send buffer happened before or after the contents of the buffer have been sent, leading to a potential incorrect transferred data.

Algorithm 2 Detecting data buffers and requests overwriting

Require: MPI initiation call mpi_i, with its matching completion calls matched_wait
Ensure: Warning are issued on the statements that can cause race conditions in the overlapping interval

```
1: procedure CHECK_RACE_COND(mpi_i, matched_wait)
2:     for all w ∈ matched_wait do
3:     for all Path p from mpi_i to w do
4:         for all Statement s ∈ p do
5:             if s writes on the request of mpi_i then
6:                 Raise warning about a request overwritting
7:             else if s reads or writes an output argument or s writes an input argument then
8:                 Raise warning about a possible buffer data race on s
```

The detection of buffer data races and request overwriting for nonblocking operations is described in algorithm 2. This algorithm is only performed once the initialization call has been successfully matched with its completion calls in order to have a properly defined overlapping window. It is based-on the use-def and def-use chains of each argument of the initialization call. The input or output nature of each argument of an MPI call is taken into account, in accordance with the interface definitions.

Every statement between the initiation point and its completion calls is then visited in a DFS fashion so that every path can be explored (lines 3 and 4). For each statement, we check if:

1) the request object is written, which also includes that its pointer is given to another function (line 5)
2) a send data buffer is written (line 7)
3) a receive data buffer is written or read (line 7)

For case 1, a warning for a potential request overwriting is issued. For cases 2 and 3, a warning for buffer misuse is issued.

As with algorithm 1, algorithm 2 can also be easily adapted for persistent operations. Checking potential data buffer misuse is the same, as their access is also forbidden between starting and completion calls. For potential request overwriting, the boundaries of the DFS should be changed to cover all paths from the initialization call to the matched freeing calls.

V. Evaluation

Algorithms 1 and 2 have been integrated in PARCOACH [1] as a compilation pass using LLVM 10. Being applied on the intermediate representation of a program, LLVM passes are independent of the source language. Besides, PARCOACH is independent of the MPI implementation used. Our analysis has been tested with `C` and `C++` codes, and should be applicable to FORTRAN codes by using the corresponding front-end and providing the adequate representation of MPI calls. The generalized dominators and post-dominators information are built upon the CFG, dominator and post-dominator trees provided by the compilation framework. The intermediate representation is in single static assignment form, and basic data dependencies information are provided such as use-def and def-use chains.

Our implementation supports all point-to-point and collective MPI nonblocking communications. We consider all kind of communication initiations, but only the `MPI_Wait` completion call. Other completion procedures are left for future work. While our implementation operates in an intraprocedural context for convenience purposes, the algorithms of section IV are suited for an interprocedural analysis. This subject will also be the matter of a future work.

A. Static Results

Our measurements were made on a platform equipped with 12 cores Intel Xeon processors with a base clock speed of 2.20GHz, and with 20Gb of memory. PARCOACH new analysis pass is applied on each translation unit and warns the developer in case of a nonblocking communication misuse. It has been validated on multiple correct and incorrect micro-benchmarks we wrote. Listings 7, 8 and 9 show three code snippets from our micro-benchmarks suite. We measured the compile-time overhead induced by our analysis compared to a compilation without the pass on two mini applications from the Mantevo project [16]: miniMD and miniFE, and 2 CORAL benchmarks : lulesh [17] and LAMMPS [18].

In Listing 7 the initiation call line 9 is not post-dominated by a completion call which means it is not correctly completed. PARCOACH successfully identifies a completion is missing and raises a warning.

The pass has limited information about the source code when compiled with debugging symbols. It provides a basic

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

Listing 7: Code snippet 1 from the micro-benchmark suite

```
7  int msg = 0;
8  MPI_Request req;
9  MPI_Ibcast(&msg, 1, MPI_INT, 0, MPI_COMM_WORLD, &req);
```

report of the encountered error, with its nature and its location in the source code. The output on the standard error stream for this example is shown below.

```
PARCOACH: warning: MPI_Ibcast line 9 has
no matching completion call
```

Listing 8: Code snippet 2 from the micro-benchmark suite

```
15 MPI_Com com = MPI_COMM_WORLD;
16 MPI_Request req;
17 MPI_Isend(&msg, 1, MPI_INT, 1, 0, com, &req);
18 if (rank < 2) {
19     printf("rank %d does nothing\n", rank);
20 } else {
21     MPI_Wait(&req, MPI_STATUS_IGNORE);
22 }
23 MPI_Wait(&req, MPI_STATUS_IGNORE);
```

Listing 8 shows an example of a misplacement of completion calls. Our pass associates the nonblocking initiation send line 17 with the `MPI_Wait` line 23 that post-dominates it and thus raises a warning for the unmatched completion line 21. The output returned by PARCOACH is

```
PARCOACH: warning: MPI_Wait line 21
is unmatched
```

The code snippet shown in Listing 9 exposes an example of a data race. `MPI_Ireduce` line 18 is matched with `MPI_Wait` line 20. Our buffer data race detection checks all instructions in the overlapping window (i.e., between lines 18 and 20). The instruction line 19 writes into `sum`. Since it is an outbound buffer, it should not be modified.

Listing 9: Code snippet 3 from the micro-benchmark suite

```
15 MPI_Com com = MPI_COMM_WORLD;
16 msg = 1+rank, sum=0;
17 MPI_Request req;
18 MPI_Ireduce(&msg, &sum, 1, MPI_INT, SUM, 0, com, &req);
19 sum = foo + rank;
20 MPI_Wait(&req, MPI_STATUS_IGNORE);
```

The warning returned by PARCOACH indicates the instruction causing a race condition and associates it with the nonblocking reduce initiation.

```
PARCOACH: warning: Race condition on
instruction line 19 - Buffer used in
MPI_Ireduce line 18
```

Table I gives an overview of the CORAL and Mantevo benchmarks that we used to evaluate our pass. The last line of the table gives the number of MPI nonblocking calls, both initiations and `MPI_Wait` calls, that were found by our analysis pass for each benchmark. The compilation-time overhead is shown in Figure 2 for those benchmarks. We omit to show the overhead of all codes in our micro benchmarks suite as it is negligible. We observe that the compilation-time overhead for lulesh is significantly larger than that of LAMMPS, despite both of them having a similar amount of MPI nonblocking calls. This difference can be explained by the smaller compilation time of lulesh. The processing of a hundred of MPI calls might induce a constant cost that can be diluted into the computation time of LAMMPS, but not lulesh.

PARCOACH new static analysis performs a more advanced data-flow study of the code than its previous analysis, which causes a higher overhead for some benchmarks. However, as seen in the table, those additional costs are still small, ranging from a couple of seconds to a few dozens for larger codes, compared to the execution time of each application. Furthermore, PARCOACH detects and reports issues in the code at an earlier point in the program lifespan, which is beneficial.

	lulesh	LAMMPS	miniMD	miniFE
Lines of code	5,000	500,000	9,000	2,000
Compiled files	5	379	12	7
Avg. Compil. time w/o pass (s)	10.920	492.072	12.601	13.069
Avg. Compil. time w/ pass (s)	16.718	581.689	14.478	14.444
Nbr. of detected NB. MPI calls	97	103	14	11

TABLE I: Statistics on tested benchmarks

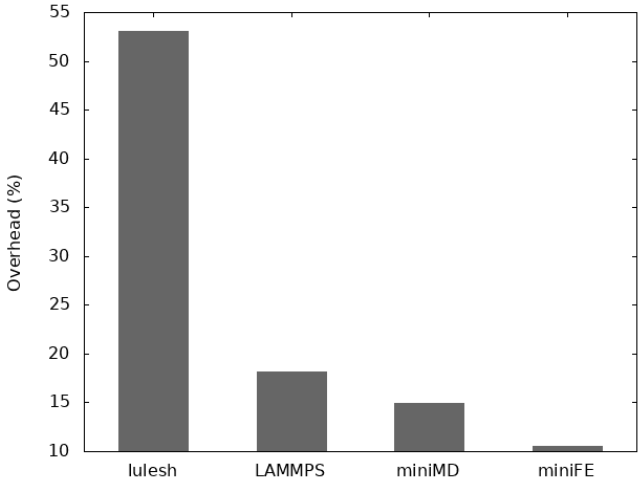

Fig. 2: Compilation-time overhead (ratio between the new PARCOACH analysis time and the total compilation time).

B. Discussion

The warnings returned by PARCOACH on the CORAL benchmarks and the mini applications are all false positives. Some situations would require a more in-depth analysis of the control and data flows. An example is given in Listing 10. The nonblocking receive initiation in the first loop must be

matched with the completion call in the second loop, since both of these loops have the same iteration domain. However our analysis is unable to match them since they do not share any domination or post-domination relationship. One way to resolve this shortcoming would be having a path-sensitive analysis.

Listing 10: Code snippet from miniFE (File `make_local_matrix.hpp`)

```
for(int i=0; i<num_recv_neighbors; ++i) {
    MPI_Irecv(&A.elements_to_send[j], send_length[i],
         mpi_dtype, neighbors[i],
              MPI_MY_TAG, MPI_COMM_WORLD, &request[i]);
    j += send_length[i];
}

/* ... */

for(int i=0; i<num_recv_neighbors; ++i) {
  if (MPI_Wait(&request[i], &status) != MPI_SUCCESS) {
    std::cerr << "MPI_Wait error\n"<<std::endl;
    MPI_Abort(MPI_COMM_WORLD, -1);
  }
}
```

Other limitations are caused by the intraprocedural context preventing us from matching calls across function boundaries and from knowing the sensitivity of the resources inside a function. The data dependency analysis is not able to properly discriminate access to structured data fields.

When `MPI_Test` is used, our analysis is able to match the initiation to the first test completion it encounters. This is a limitation of our analysis as the operation may not be completed at the first test. However only a dynamic analysis could improve this solution.

The support for other flavors of completion calls requires more extensive work. The `all` versions need a more precise data flow analysis that is able to detect array aliases. The `some` and `any` would require a run-time analysis to determine which requests have been caught.

VI. Conclusion

In this paper we present an extension of PARCOACH to detect misuse of MPI nonblocking and persistent communications. We propose two algorithms based on the notion of generalized dominators and post-dominators to add new error detection in PARCOACH. Our analysis is built on top of LLVM 10 and can automatically find five type of errors. In addition to the already implemented detection of collective mismatch, which was augmented to also include persistent initialization calls, the following correctness analyses are now possible with PARCOACH: missing wait, unmatched wait, request overwriting and buffer data race. This analysis can be easily coupled with any optimization pass (e.g., [19]) to verify nonblocking communication transformations.

For future work, we plan on addressing the following limitations in our implementation. In its current state, it only supports `MPI_Wait` and `MPI_Test` as completion procedures. We plan to extend this list to all their flavors (`MPI_Wait/Test{all/any/some}`) which require a more precise, element-wise analysis and an adaptation of PARCOACH's existing dynamic verification to catch those situations. The current implementation of this new analysis is also limited to intraprocedural. This is an issue when dealing with programs using wrappers around the MPI calls. To support these cases, we plan to adapt our analysis to the existing interprocedural verification in PARCOACH [2]. Finally, the support for persistent operations is preliminary. The algorithm can be improved to perform all the matching in one pass instead of performing the algorithm multiple times.

References

[1] J. Jaeger, E. Saillard, P. Carribault, and D. Barthou, "Correctness Analysis of MPI-3 Non-Blocking Communications in PARCOACH," in *European MPI Users' Group Meeting* , ser. EuroMPI '15 The 22nd European MPI Users' Group Meeting, Bordeaux, France, Sep. 2015.

[2] P. Huchant, E. Saillard, D. Barthou, H. Brunie, and P. Carribault, "Parcoach extension for a full-interprocedural collectives verification," in *2018 IEEE/ACM 2nd International Workshop on Software Correctness for HPC Applications (Correctness)*. IEEE, 2018, pp. 69–76.

[3] P. Huchant, E. Saillard, D. Barthou, and P. Carribault, "Multi-valued expression analysis for collective checking," in *Euro-Par 2019: Parallel Processing*, 2019, pp. 29–43.

[4] M. P. I. Forum, *MPI: A Message-passing Interface Standard, Version 3.1 ; June 4, 2015*, 2015. [Online]. Available: https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

[5] 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.

[6] F. Ye, J. Zhao, and V. Sarkar, "Detecting mpi usage anomalies via partial program symbolic execution," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC 18. IEEE Press, 2018. [Online]. Available: https://doi.org/10.1109/SC.2018.00066

[7] A. Vo, S. Aananthakrishnan, G. Gopalakrishnan, B. R. d. Supinski, M. Schulz, and G. Bronevetsky, "A scalable and distributed dynamic formal verifier for mpi programs," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC 10. USA: IEEE Computer Society, 2010, p. 110. [Online]. Available: https://doi.org/10.1109/SC.2010.7

[8] A. Pham, T. Jéron, and M. Quinson, "Verifying mpi applications with simgridmc," in *Proceedings of the First International Workshop on Software Correctness for HPC Applications*, ser. Correctness17. New York, NY, USA: Association for Computing Machinery, 2017, p. 2833. [Online]. Available: https://doi.org/10.1145/3145344.3145345

[9] A. Droste, M. Kuhn, and T. Ludwig, "Mpi-checker: Static analysis for mpi," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, 2015, pp. 1–10.

[10] Z. Luo, M. Zheng, and S. F. Siegel, "Verification of mpi programs using civl," in *Proceedings of the 24th European MPI Users Group Meeting*, ser. EuroMPI 17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3127024.3127032

[11] H. Yu, "Combining symbolic execution and model checking to verify mpi programs," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE 18. New York, NY, USA: Association for Computing Machinery, 2018, p. 527530. [Online]. Available: https://doi.org/10.1145/3183440.3190336

[12] H. Yu, Z. Chen, X. Fu, J. Wang, Z. Su, J. Sun, C. Huang, and W. Dong, "Symbolic verification of message passing interface programs," ser. ICSE 20, 2020.

[13] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "An efficient method of computing static single assignment form," in *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL 89. New York, NY, USA: Association for Computing Machinery, 1989, p. 2535. [Online]. Available: https://doi.org/10.1145/75277.75280

[14] R. Gupta, "Generalized dominators and post-dominators," in *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1992, pp. 246–257.

[15] V. C. Sreedhar, G. R. Gao, and Y. Lee, "Dj-graphs and their applications to flowgraph analyses," in *ACAPS Tech. Memo 70*. Citeseer, 1994.
[16] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving Performance via Mini-Applications," *Sandia National Laboratories, Tech. Rep. SAND2009-5574*, vol. 3, 2009.
[17] I. Karlin, J. Keasler, and J. Neely, "Lulesh 2.0 updates and changes," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2013.
[18] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *Journal of computational physics*, vol. 117, no. 1, pp. 1–19, 1995. [Online]. Available: http://lammps.sandia.gov
[19] V. M. Nguyen, E. Saillard, J. Jeager, D. Barthou, and P. Carribault, "Automatic code motion to extend mpi nonblocking overlap window," 2020.