PARCOACH: Combining static and dynamic validation of MPI collective communications

Emmanuelle Saillard, Patrick Carribault, Denis Barthou

PARCOACH: Combining Static and Dynamic Validation of MPI Collective Communications

Emmanuelle Saillard[*]*, Patrick Carribault*[*]*and Denis Barthou*[†]

Abstract

Nowadays most scientific applications are parallelized based on MPI communications. Collective MPI communications have to be executed in the same order by all processes in their communicator and the same number of times, otherwise it is not conforming to the standard and a deadlock or other undefined behavior can occur. As soon as the control-flow involving these collective operations becomes more complex, in particular including conditionals on process ranks, ensuring the correction of such code is error-prone. We propose in this paper a static analysis to detect when such situation occurs, combined with a code transformation that prevents from deadlocking. We focus on blocking MPI collective operations in SPMD applications, assuming MPI calls are not nested in multithreaded regions. We show on several benchmarks the small impact on performance and the ease of integration of our techniques in the development process.

Keywords: MPI, debugging, collective, static analysis, correctness

1 Introduction

Most of scientific applications in High-Performance Computing rely on the MPI API specification to efficiently exploit a supercomputer and reach high parallel performance. Based on the distributed-memory paradigm, this model exposes multiple ways to express communications between tasks/processes including point-to-point and collective. While point-to-point functions involve only two tasks, collective communications require that all processes in a communicator invoke the same operation. Each process does not have to statically invoke such collective function at the same line of the source code, but the sequences of collective calls in all MPI processes must be the same and corresponding function calls should have a compatible set of arguments. Due to the control flow inside an MPI program, processes may execute different execution paths. Such behavior may cause errors and deadlocks difficult for the user to detect and analyze. To tackle this issue, this paper presents PARCOACH (PARallel COntrol flow Anomaly CHecker) a two-step analysis to detect incorrect collective patterns in SPMD (Single Program Multiple Data) MPI programs. For each function, we first identify at compile-time the code fragments calling collectives that may deadlock and the control-flow parts that may lead to such situation. Warnings are issued during this phase. Then, we transform the identified code fragments in order to dynamically capture these situations before they arise. The runtime overhead of this instrumentation is limited since only the pieces of code calling collectives that can deadlock are modified. In case of actual misuse, the application stops with an explicit error message highlighting both the collectives and the control-flow code responsible for this situation. This paper is an extension

[*]CEA, DAM, DIF, F-91297 Arpajon, France
[†]Bordeaux Institute of Technology, LaBRI / INRIA, Bordeaux, France

of [16] including new detailed examples, new experimental results and a study about the check collective function (*CC*) used for the code transformation.

1.1 Motivating Example

The following simple example (Listing 1) illustrates the potential issues with collective communications.

Listing 1: A simple example

```
void f( int r ) {
   if( r == 0 )
      MPI_Barrier(MPI_COMM_WORLD);
   return;
}
void g( int r ) {
   f(r);
   MPI_Barrier(MPI_COMM_WORLD);
   exit(0);
}
```

Listing 2: The instrumented example

```
void f( int r ) {
   int res;
   if( r == 0 ) {
      MPI_Reduce(1,&res,1,MPI_INT,equalsop,0,MPI_COMM_WORLD);
      if ( rank==0 && res == -1 ) MPI_Abort(MPI_COMM_WORLD,0);
      MPI_Barrier(MPI_COMM_WORLD);
   }
   MPI_Reduce(0,&res,1,MPI_INT,equalsop,0,MPI_COMM_WORLD);
   if ( rank==0 && res == -1 ) MPI_Abort(MPI_COMM_WORLD,0);
   return;
}
```

Assume here that `g` is called by all processes. Depending on the value of the input parameter `r`, a process will execute or not the barrier in the `if` statement in `f`. If `r` is not uniformly true or false among MPI processes, some tasks will be blocked in `f` while the remaining process ranks will reach the barrier in `g`. These processes will then terminate, while the first ones will be in a deadlock situation at the barrier in `g`. The machine state when the deadlock occurs does not help to identify the cause of the deadlock. As the value of `r` is unknown at compile time and might be the same for every MPI process, the dynamic state of control flow has to be checked in order to prevent from entering a deadlock state. Transforming the previous example would lead to the code presented in Listing 2. Notice that the function g does not need to be transformed since it does not introduce a collective that may be the cause of a deadlock or an unspecified behavior. In order to partition processes according to their behavior regarding the conditional in `f`, two calls to the collective `MPI_Reduce` with the equalsop operation (bit equality checking, see Section 3.2) are inserted in the code: One before the barrier operation with the input value 1 (1^{st} parameter of the call), and one before the `return` statement with the input value 0. All processes call the `MPI_Reduce` collective, whatever their execution path. However, input values should be the same, otherwise the function is incorrect and `MPI_Abort` is issued in order to prevent from deadlocking.

We propose in this paper methods (i) to identify the conditional in `f` as the cause for a possible deadlock, using compiler analysis and (ii) to prevent from deadlocking using a code transformation.

1.2 Context and Contributions

This paper focuses on scientific SPMD (Single Program Multiple Data) applications parallelized with MPI, meaning that the source code functions are assumed to be called either by all processes or by none of them. Futhermore, we consider only monothreaded MPI programs (the analysis also works on multithreaded programs if all MPI collectives are performed in monothreaded regions). In our context a function is said to be *correct* regarding blocking collective communications if all MPI processes entering the function eventually exit without leaving any process blocked inside a collective operation. As our analysis is focused on the detection of mismatching collectives, other possible sources of deadlock (e.g., infinite loops, blocking IOs and other deadlocks) which would

require dedicated analysis are not checked. While all types of blocking collectives are handled, collective operations are assumed to be called on the same communicators, with compatible arguments. In this context, this article makes the following contributions:

- Identification of collective callsites that may lead to deadlocks, and of control-flow codes responsible for such situation within each function.
- Dedicated instrumentation based on the previous analysis to prevent collective errors at execution time, pinpointing the control-flow divergence responsible for such errors.
- Full implementation inside a production compiler, experimental results on MPI benchmarks and applications.

1.3 Outline

Section 2 describes related work on MPI debugging and analysis, focusing on collective operations. Section 3 presents our contribution: the static analysis detecting collective issues and the code transformation to capture incorrect functions. Section 4 shows experimental results and finally Section 5 concludes our work.

2 Related Work

Related work on MPI code verification can be organized in 3 categories: (i) static analyses, (ii) online dynamic analyses and (iii) trace-based dynamic analyses.

Static tools. This class of tools is mainly based on model checking and requires symbolic program execution, at the expense of combinatorial number of schedules or reachable states to consider. TASS[18], a successor of MPI-SPIN[19] follows this approach: using model checking and symbolic execution, it checks numerous program properties explicitly annotated with pragmas. If a property is violated (such as an incorrect order of collective calls) by exploring reachable states of the model built, an explicit counter-example is returned to the user in the form of a step-by-step trace through the program showing the values of variables at each state. Unlike TASS, our static check analyzer requires no source-code modifications since it is integrated within a compiler. Potential errors are automatically returned to the programmer with their context (including the line of the erroneous conditional) through a low-complexity control-flow graph analysis. However a pragma-based approach could be useful to improve our static analysis (for example by tagging MPI rank dependent variables), thus reducing false-positive possibilities. Besides, in our approach, the combinatorial aspect of detecting effective mismatch is avoided by the runtime check.

Online dynamic tools. Dealing with dynamic tools able to check collective operations, we can mention DAMPI[22], Marmot[13, 17], Umpire[21, 17], MPI-CHECK[14, 17], Intel Message Checker (IMC)[6, 17] and MUST[11, 10]. Umpire, Marmot and MUST rely on a dynamic analysis of MPI calls instrumented through the MPI profiling interface (PMPI). They are able to detect mismatching collectives either with a timeout approach (DAMPI, Marmot, IMC and MPI-CHECK) or with a scheduling validation (Umpire and MUST). Methods performing deadlock detections through a timeout approach are known to produce false positives, for example in case of abnormal latencies. DAMPI uses a scalable algorithm based on Lamport Clocks (vector clocks focused on call order) to capture possible non deterministic matches. For each MPI collective operation, participating processes update their clock, based on operation semantics. Umpire, limited to shared memory platforms, relies on dependency graphs with additional arcs for collective operations to detect deadlocks. In Marmot, an additional MPI process performs a global analysis of function calls and communication patterns. Both of these approaches, however, have limited scalability, forwarding MPI call information to a central manager for collective correctness. MUST overcomes such limitation by relying on a tree-based layout[9]. Finally,

MPI-CHECK[14] instruments the source code at compile time adding extra arguments to MPI calls. The resulting instrumented program is then compiled and produces an instrumented executable which outputs errors and warnings upon execution. In our approach, we perform a runtime check, taking advantage of the compile time analysis results (code locus and potential error filtering) in order to scale to large programs, avoiding instrumentation of the whole MPI interface or systematic code instrumentation like in MPI-CHECK.

Validation can also be done inside MPI libraries or as an extension of a library (as for MPICH for instance), allowing collective verification for the full MPI-2 standard[7, 8, 20]. The detection of runtime deadlock causes is however limited to the information available to the MPI routines. Compared to other dynamic analysis tools, our method provides more precise errors including the conditionals responsible with our static check help.

Trace-based dynamic tools. IMC (recently replaced by the online tool Intel Trace Analyzer v9.0) collects all MPI-related information in trace files and performs the post-mortem analysis of these traces. This tends to be difficult and with limited scalability due to the trace sizes, correlated to the number of cores and execution time of the application.

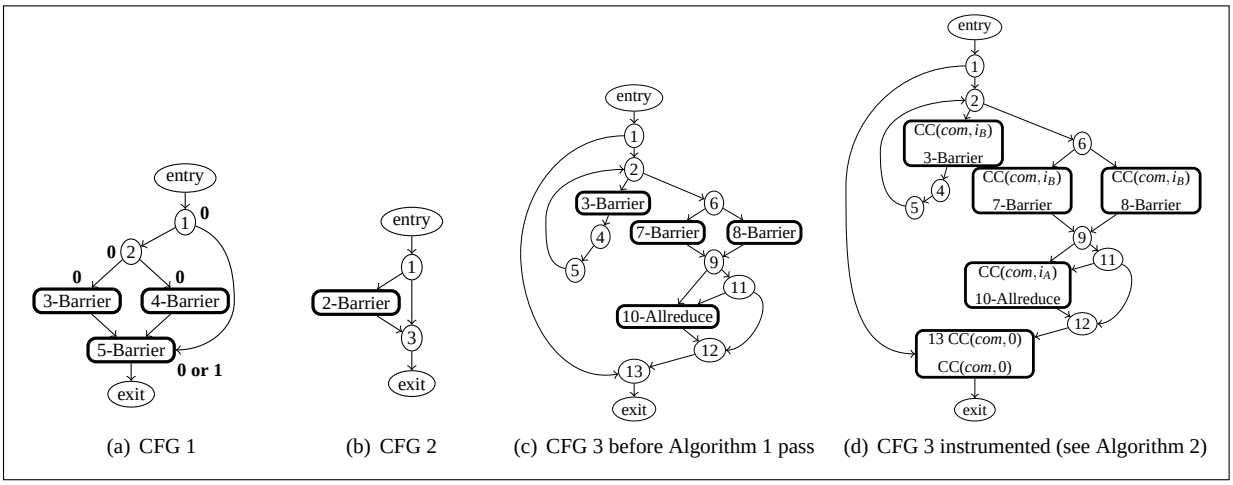
Our detection of incorrect functions combines both static and dynamic approaches. The static analysis detects all incorrect functions of a program and issues warnings for potential errors. Then because these potential errors might not appear during execution, the code is transformed in order to check only the reported warnings. In case of actual deadlock situation, the program aborts allowing a program state exploration with a debugger. PARCOACH checks a program function by function and then can stop the program before all existing dynamic tools. As our dynamic check is performed by a lightweight library (see Algorithm 3), it is also independent from the MPI implementation.

3 Checking MPI Collective Operations

This section describes our combining method named PARCOACH to verify MPI programs. To prove an MPI program is correct, PARCOACH decomposes the method into two phases: A compile-time verification and an execution-time verification. All source code functions are either proved statically correct, with the meaning given in the introduction, or potentially incorrect, depending on the control flow. Correct functions are filtered out and the code of the remaining functions is transformed to prevent deadlock situations. Only collective calls that can deadlock are instrumented. This filtering approach avoid systematic instrumentation, thus reducing the overhead due to the dynamic analysis. When a deadlock situation occurs in a run, an error message is returned with information gathered at compile-time: The location and the type of the collective and the control-flow code responsible for this situation.

3.1 Compile-Time Analysis

The first step takes place in the middle of the compilation chain where the source code is represented in an intermediate form. This step consists in a static analysis of the *control-flow graph (CFG)* for each function. The CFG is defined as a directed graph (V,E) where V represents the set of basic blocks and E is the set of edges. Each edge $u \longrightarrow v \in E$ depicts a potential flow of control from node u to v. Each node in V has a set of successors denoted as $SUCC(u)$. Moreover, we assume that the underlying compiler appends two unique artificial nodes for entry and exit points. Throughout the rest of the paper, node refers to a CFG node. Within the CFG, the different paths correspond to possible execution paths that may be taken by the different MPI tasks. The principle of the proposed static analysis is to detect functions that have paths with different sequences of collectives (either not the same number or not the same collectives). When two such paths are found, the node responsible for this possible control flow divergence leading



(a) CFG 1 (b) CFG 2 (c) CFG 3 before Algorithm 1 pass (d) CFG 3 instrumented (see Algorithm 2)

Fig. 1: Example of Control Flow Graphs. From the left, a CFG showing execution orders, CFG of function f, a CFG from a Benchmark and its instrumentation (see Algorithm 2)

to deadlock is identified. Algorithm 1 details our compile-time analysis to detect if a function is correct (see Section 1.2). The algorithm takes as input the CFG of the current function and outputs nodes that may lead to collective errors and their collectives that may deadlock (set O). This set will be given as parameter to the instrumentation detailed in Section 3.2. The main steps of the algorithm are the following: we compute for each node of the CFG the number of collectives on the execution paths from the function entry to the node. This number is 0 for nodes before the first collective (including the node with the first collective), 1 for nodes reached after one collective and so on. When multiple paths exist, nodes can have multiple numbers, at most the number of collectives in the function. Loop backedges are removed to have a finite numbering and the algorithm is applied to the CFG of each loop separately. For nodes with collectives, these numbers define an *execution order* between collectives within a function. Collectives with different numbers are executed sequentially while directives with the same number can be executed in parallel. These numbers are called in the following *execution orders*. If the same node has multiple execution orders, only the highest one is considered. In a correct function, for any given order k, all execution paths from *entry* to *exit* should traverse the nodes of order k with the same collective operation. A function is not correct if there are paths traversing nodes of execution order k and other paths that do not traverse nodes of order k or with different collectives. They can be computed using the iterated postdominance frontier [5]. A node u postdominates a node v if all paths from v to *exit* go through u. We extend this relation to sets: A set U postdominates a node v if all paths from v to *exit* go through at least one node of U. The postdominance frontier of a node u, $PDF(u)$ is the set of all nodes v such that u postdominates a successor of v but does not strictly postdominate v. If $\gg$ denotes the postdominance relation, $PDF(u) = \{v \mid \exists\, w \in SUCC(v), u \gg w \text{ and } u \not\gg v\}$. In other words all paths from w to the exit node go through u. On the contrary v is not postdominated by u so there exists a path from v to exit node that does not traverse u. This notion is extended to a set of nodes U. The iterated postdominance frontier PDF^+ is defined as the transitive closure of PDF, when considered as a relation [5].

Algorithm 1 describes this computation applied to each function and loop, entry and exit being then defined as loop entry and exit. The execution order computation corresponds to a simple traversal of the acyclic CFG, counting traversed nodes with collectives. Then for each execution order r, the nodes calling the same function c, at order r are clustered into $C_{r,c}$. The iterated postdominance frontier of this set corresponds to nodes that can lead both to the execution of such collective or not. Note that the

Algorithm 1 Step 1 - Static Pass

```
1: function STATIC_PASS(G = (V,E))                                   ▷ G: CFG
2:     O ← ∅                                                         ▷ Output set
3:     Remove loop backedges in G to compute execution orders for nodes with collectives
4:     for r in node orders do
5:         for c in collective names of execution order r do
6:             C_{r,c} ← {u ∈ V | order r is the max. execution order of u, u executes a collective with name c}
7:             if PDF+(C_{r,c}) ≠ ∅ then
8:                 O ← O ∪ (c, PDF+(C_{r,c}))
9:             end if
10:        end for
11:    end for
12:    Output nodes in O as warnings and for Step 2.
13: end function
```

algorithm can handle any collective operation, and based on our context, only the name of the collective is used in the algorithm.

Lemma 1. *Algorithm 1 is correct if it detects all deadlock situations due to an MPI collective operation.*

Proof. We prove that the algorithm computes a non-empty set O if and only if the function is incorrect, and nodes in O correspond exactly to the nodes that can lead to a deadlock.

Consider an element (c,S) of O, with c a collective and S the set $PDF^+(C_{r,c})$ for some order r. If u denotes a node from S, there is an outgoing path from u that goes through c of order r, and another path that reaches the exit node without going through a collective c of same order. If the second path never reaches a collective c (any order) and if both paths are executed by different tasks, then some tasks will wait at the collective c while the other tasks will either wait at another collective (a deadlock) or exit the function (incorrect function). In both cases, the function is incorrect. If both paths traverse the same collective c, since the orders are different, one of the paths has more collectives c than the other. Again, this leads to an incorrect function.

The algorithm is applied on each loop separately. This separate analysis identifies at least loop exit nodes as control-flow nodes that may be responsible for deadlocks, when the loop calls collectives. Indeed, static analysis does not count iterations and collectives in loops may be executed a different number of times for each process.

Now consider an incorrect function: when executing this function with multiple MPI tasks, some tasks may reach the exit node while other tasks are waiting at a collective c inside the function. If this collective is not inside a loop, by definition of the execution order, this implies that the exit node and the node with the collective c have the same execution order r. As nodes may have multiple execution orders, let us consider the smallest r. There is a collective c' of order r such that the set of nodes $C_{r,c'}$ is not empty. Besides, $PDF^+(C_{r,c'}) \neq \emptyset$ since there is a path to the exit that does not traverse the r^{th} collective. Hence Algorithm 1 detects the function as incorrect. A similar proof holds for the case where the tasks are executing two different type of collectives. □

Figure 1(a) shows an example of execution orders (numbers near each node) computation for a simple CFG. `MPI_Barrier` node 5 can be the first collective called (path *entry*→1→5) or the second one (path *entry*→1→2→3 or 4→ 5). On this CFG, the set of nodes {3, 4} postdominates node 2: $\{3,4\} \gg 2$ but $\{3,4\} \not\gg 1$ so node 1 is in the iterated postdominance frontier of the set of nodes {3, 4}: $PDF^+(\{3,4\}) = \{1\}$. Figure 1(b) depicts the CFG extracted from the initial code of Section 1.1. It contains 3 nodes: The first one represents the `if` statement while the second one contains the `if` body with the collective call. Finally the last one denotes the `return` instruction. The algorithm considers the set $C_{0,\texttt{Barrier}} = \{2\}$ corresponding to the collective `MPI_Barrier`. As its iterated postdominance frontier is node 1, the algorithm outputs a warning for the condition located in node 1 and flags

the collective MPI_Barrier for the following dynamic analysis (set O). Figure 1(c) presents another CFG extracted from a real benchmark. This example contains 2 collectives: MPI_Barrier (nodes 3, 7 and 8) and MPI_Allreduce (node 10). The algorithm first removes the backedge $5 \rightarrow 2$ from the loop and computes orders. Nodes $7,8$ are of order 0, 10 of order 1. For the collectives in $C_{0,\texttt{Barrier}} = \{7,8\}$, the iterated postdominance frontier corresponds to node 1. Note that node 6 is postdominated by the set $\{7,8\}$ according to the definition of previous section. $C_{1,\texttt{Allreduce}}$ contains only node 10 and $PDF^+(C_{1,\texttt{Allreduce}}) = \{1,9,11\}$. Indeed from these nodes, it is possible to execute the MPI_Allreduce or not. Finally, the same algorithm is applied once more on the graph with nodes $\{2,3,4,5\}$ corresponding to the loop, without the backedge. Node 2 is marked as entry and exit. This node is the only one in the iterated postdominance frontier of the barrier in node 3. To sum up, node 1 decides of the number of execution of barriers in $7,8$, nodes $9,11$ decide of the number of execution of MPI_Allreduce and node 2 is responsible for the number of barriers executed in node 3.

Potential errors reported by the static analysis can be false positives relatively to the CFG that is not correlated to the actual control flow. To deal with false positive a dynamic check is needed.

3.2 Static Instrumentation for Execution-Time Verification

The code fragments leading potentially to incorrect functions and detected with the previous analysis are transformed in order to raise an error message at the execution time: Whenever MPI processes take execution paths that cannot lead to the same number of collectives, in the same order, the program stops. This section presents the code transformation involved.

Some potential errors may depend on the control flow taken by the different processes. The main idea is to modify the code so that before each MPI collective call, we check that all processes within the communicator are about to call the same collective. Besides, we also check that when a process is going to exit the function, all processes are exiting. This is achieved by a function, CC that counts the number of processes that are going to execute a given collective operation or to exit the function in which the MPI collective operation is invoked. CC is also a collective operation, as it gathers the processes of the communicator into groups depending on what they are going to call (collective type or exit). Function CC is depicted in Algorithm 3. It takes as input the communicator related to the collective call c, an integer i_c identifying the type of collective and the set of nodes generated by the previous algorithm (see Section 3.1). We define a new MPI operator named *equalsop* which returns -1 if there is at least two different integer among processes. Relying on the CC function, Algorithm 2 describes the instrumentation for the execution-time verification. The function INSTRUMENTATION is called on MPI_COMM_WORLD. For each node n containing a call to the collective c, MPI_Reduce is called just before calling c. The root process provides the combined value and test if all processes have the same input value. If input values are different among all processes, an error is issued and the program is aborted through a call to MPI_Abort. This process is repeated for each collective operation c in the set O. Finally, in the closest node of collective nodes that postdominates and joins all paths of the CFG, MPI_Reduce with the input value 0 is added to eventually catch up processes not calling any additional collective. Figure 1(d) presents the transformation achieved by Algorithm 2 on the CFG Figure 1(c).

Algorithm 2 Step 2 - Selective Static Instrumentation

```
1: function INSTRUMENTATION(communicator, G, O)        ▷ G: CFG, O: set created by Algorithm 1
2:     for (c, S) ∈ O do
3:         for n in nodes containing a call to collective c do
4:             Insert call to CC(communicator, i_c, S) before the call to c
5:         end for
6:     end for
7:     Insert call to CC(communicator, 0, ∅) before return statements
8: end function
```

Algorithm 3 Library Function To Check Collectives (CC)

```
1: function CC(communicator, i_c, S)
2:     int rank, res
3:     MPI_COMM_RANK(communicator, &rank)
4:     MPI_REDUCE(&i_c, &res, 1, MPI_INT, equalsop, 0, communicator)
5:     if rank == 0 && res == −1 then
6:         Display error for all nodes in S
7:         MPI_ABORT(communicator, 0)
8:     end if
9: end function
```

Lemma 2. *Algorithm 2 is correct if all deadlock situations are captured by the instrumentation and if the new collectives inserted do not generate a deadlock themselves.*

Proof. We define a *control sequence* as the sequence of collective calls executed by a process in a program execution. For an execution of a given function, a control sequence is denoted as $c_1c_2..c_n$ with c_i the i-th collective called. Algorithm 2 rewrites each collective c_j from the set O into s_jc_j corresponding to the function `MPI_Reduce` called by *CC* based on the color j and the initial collective c_j. The function `MPI_Reduce` with color 0 denoted as s_0 is added after all collective nodes. To ease the proof, we will assume that this conditional rewriting, performed only for collectives found by the static analysis, is conducted for all collectives of the control sequence. Consequently, a sequence $c_1..c_n$ becomes $s_1c_1..s_nc_ns_0$. If all control sequences are the same for all processes, the function executes with no deadlock. By applying Algorithm 2, the modified control sequences are still identical, this algorithm does not introduce deadlocks. If a function deadlocks due to collective operations,

- Either a process calls a collective communication c_i while another process calls a collective function c_k with $k \neq i$. The control sequence of both processes differ only with their last collective, c_k and c_i, and both are prefixed by $c_1..c_{i-1}$.
- Or a process calls a collective communication while another one exits the function (a deadlock may occur at a later point in the execution or outside of the function). The control sequence of the process exiting the function is $c_1..c_{i-1}$ and the process inside the function executes the same prefix sequence with one more collective c_i.

In the first case, the algorithm changes both control sequences into $s_1c_1..s_{i-1}c_{i-1}s_i$ and $s_1c_1..s_{i-1}c_{i-1}s_k$. These sequences stop with s_i and s_k since CC(x,i) and CC(x,k) lead to an error detection and abort. Hence the modified function no longer deadlocks. In the second case, the algorithm changes both control sequences into $s_1c_1..s_{i-1}c_{i-1}s_i$ for the process inside the function, and $s_1c_1..s_{i-1}c_{i-1}s_0$ for the one trying to leave the function. Note that the process is stopped before leaving the function since CC(x,i) and CC(x,0) both abort, generating an error message. Again, the modified function does not deadlock anymore.

To conclude, Algorithm 2 is indeed correct and prevents all deadlock situations. □

4 Experimental Results

We implemented our analysis in GCC 4.7.0[1] as a plugin performing a new pass inserted inside the compiler pass manager, after generating the CFG information. This pass is located in the middle end of the compilation chain and is written in GIMPLE[15], a tree-address representation derived from GENERIC. Thus, this solution is language independent, allowing to check MPI applications written in C, C++ or FORTRAN (MPI collective operations are identified by their name). Besides the application language is known at compile-time which enables the *CC* function to be adapted to all applications. The pass applies Algorithms 1 and 2. The application needs to be linked to our dynamic library for runtime checking (see Algorithm 3). Our static analysis is simple to deploy

in existing environment as it does not modify the whole compilation chain. This section presents experimental results obtained on representative C++ MPI applications: EulerMHD[23], solving the Euler and ideal magnetohydrodynamics equations both at high order on a 2D Cartesian mesh and HERA[12], a large multi-physics AMR hydrocode platform. We also selected six benchmarks from the MPI NAS Parallel benchmarks[3] (NASPB v3.2) using class C to test both C and Fortran programs.

All experiments were conducted on Tera 100, *a supercomputer* with an aggregate peak performance of 1.2 PetaFlops. It hosts 4,370 compute nodes for a total of 140,000 cores. Each compute node gathers four eight-core Nehalem EX processors at 2.27 GHz and 64 GB of RAM. All performance results are computed as the average over 8 runs (compilation or execution) with BullxMPI 1.1.14.3 and Linux version 2.6.32.

4.1 CC function implementation

The goal of the `CC` function that checks MPI collective operations at runtime is to gather MPI processes calling the same collective call. For that purpose, we can split the communicator through a call to `MPI_Comm_split`, gather or reduce information about collectives with `MPI_Allgather`, `MPI_Gather`, `MPI_Allreduce` or `MPI_Reduce`, the best approach depending on the implementation. To determine which MPI function performs better, we used the Intel MPI Benchmark suite[2] (IMB) v3.2.3 with a 4B message as we only want to send an integer related to the type of the collective about to be called. Figure 2 shows the time spent in each candidate function for a range of MPI processes. In this figure, `MPI_Reduce` seems to be the most scalable[4]. Hence we opted for this function.

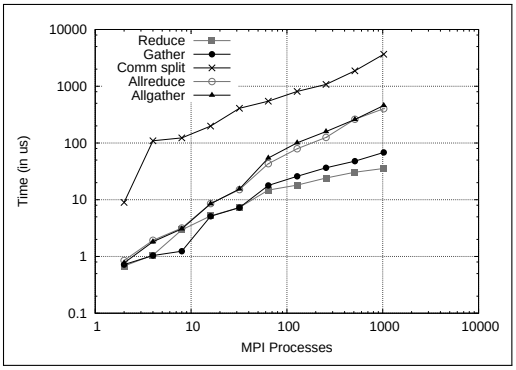


Fig. 2: Execution time of collective calls from IMB

Benchmark	#coll. calls	#nodes in *S*	% instrumented collectives	#calls to CC
EulerMHD	14	14	36%	26
BT	10	5	78%	8
LU	16	2	14%	6
SP	9	5	75%	7
IS	5	2	40%	3
CG	2	0	0%	0
FT	10	0	0%	0
HERA	644	578	84%	3,255

Tab. 1: Compilation and Execution Results

4.2 Static Check Results

At compile time, a warning is returned to the programmer when a potential deadlock situation is detected. The following example shows what a user can read on `stderr` for NAS benchmark IS:

```
is.c:In function ‘main’:
is.c:1093:1: warning: STATIC-CHECK: MPI_Reduce may not be called by all processes in the communicator
because of the conditional line 923 - Check inserted before MPI_Reduce line 994
```

This warning provides the name of the collective that may deadlock (`MPI_Reduce`) and the line of the conditional leading to the collective call (line 923). This collective call is instrumented at line 994 as described in Algorithm 2. In this case, a test over the number of processes which if higher than 512 produces an error requesting a lower number of processes, before finalizing the program. This particular warning does not lead to a deadlock as all processes inside the same communicator share the same value

for `MPI_Comm_size`. However, notice that the line number where the control flow divergence may occur is not close to the collective call: The conditional that may be responsible for a deadlock in a `MPI_Reduce` is 71 lines far from the collective.

Figure 3(a) details the overhead of compilation time when activating our GCC plugin. This overhead remains acceptable as it does not exceed 5% for HERA. It is presented with and without the code generation which accounts for the insertion of `CC` function calls (see Algorithm 3). This specific step is mainly responsible for the overhead except for CG and FT. Indeed, according to the static analysis, these benchmarks are correct, so no collective operation is instrumented. For each benchmark, Table 1 presents the number of static calls to a collective communication and the number of nodes found by Algorithm 1 (set $S = \cup(PDF^+(C_{r,c}) \in O)$). The location of the static analysis in the compilation chain explains the high number of collective calls found in HERA. Indeed, C++ templates are instantiated and, therefore, duplicated before entering the middle-end part of GCC. For all nodes in S, the control-flow does not depend on process ranks and the functions are correct. Nevertheless, this table shows that the static analysis is able to reduce the amount of instrumentation needed to check the collective patterns (third column). Reducing further the number of instrumented collectives would require an inter-procedural data-flow analysis on the nodes in S. Such analysis is outside the scope of this paper and is left for future work.

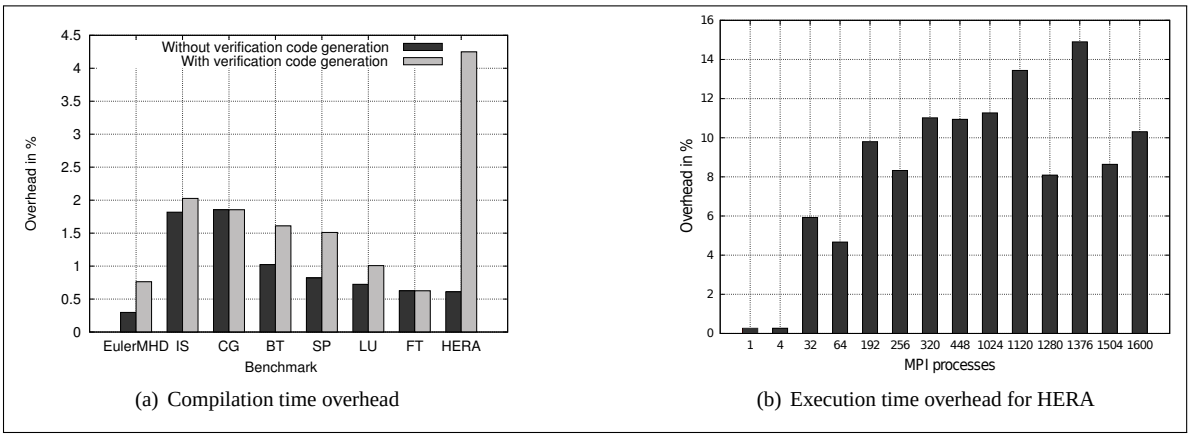
4.3 Execution Results

Figure 4(a) shows the overhead obtained for NASPB class C from 4 to 512 cores (CG and FT have no overhead as no collective is instrumented). The overhead does not exceed 18% and tends to slightly increase with the number of cores. Figure 4(b) presents weak-scaling results for EulerMHD from 1 to 1,280 cores where the overhead remains comparable with a higher overhead as it is related to the number of `CC` calls, with the same increasing trend. Figure 3(b) presents the overhead obtained for HERA from 1 to 384 cores. The overhead also increases with the number of processes and does not exceed 12%. Highest execution times are of the order of 10 min for the benchmarks. The last column of table 1 depicts the number of calls to the `CC` function during the execution of the benchmarks. Processes about to call collectives identified as potential deadlock sources are counted. If some processes are missing, the abort function is called to stop the program before deadlocking. An error is printed to `stderr` with the line number, the collective name and conditionals responsible (informations gathered at compile-time):

```
DYNAMIC-CHECK: Error detected on rank 0 - Abort is invoking before MPI_Barrier line 47 in function f (program.c)
DYNAMIC-CHECK: See warnings about conditional(s) line(s) 45
```

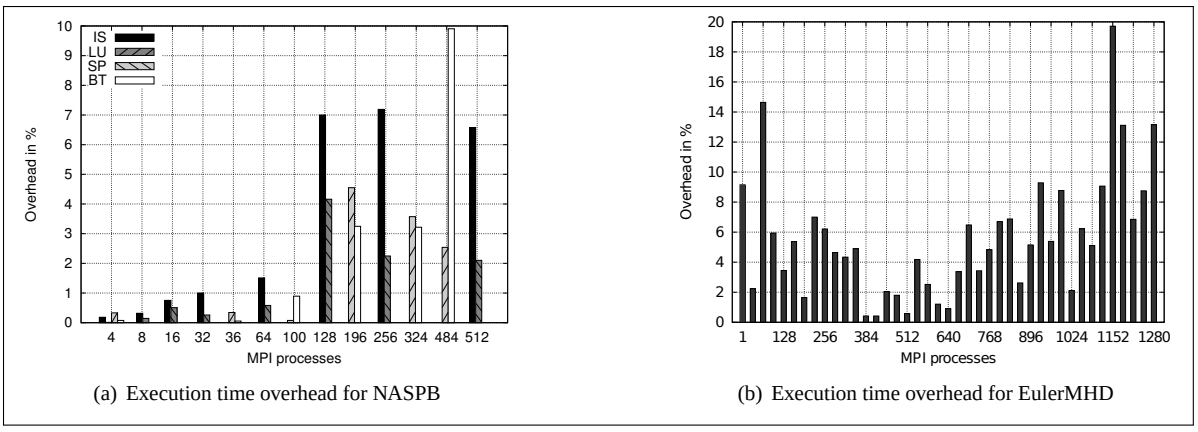
5 Conclusion and Future Work

In this paper we described PARCOACH, our two-phase analysis to detect incorrect collective patterns in MPI programs. The first pass statically identifies the reduced set of collective communications that may eventually lead to potential deadlock situations, and issues warnings. Using this analysis, a selective instrumentation of the code is achieved, displaying an error, synchronously interrupting all processes, if the schedule leads to a deadlock situation. This method is easily integrated in the GCC compiler as a plugin, avoiding compiler recompilation. We have shown that the overhead is very low at compile-time (5%). Dealing with the runtime overhead, it could become non-negligible at larger scale as our analysis adds collectives for instrumentation. However, with the help of collective selection, the runtime overhead remains acceptable (less than 20%) at a representative scale on a C++ application. Although it satisfies both scalability and functional requirements, our analysis tool is only intra-procedural with the possible drawback of missing conditional statements out of function boundaries. Moreover, our analysis is focused on a particular error and should be extended to cover common verification, for example, MPI call arguments, such as different communicators. These improvements are currently under development, and the analysis is being extended to inter-procedural analysis, gathering more data-flow information at compile-time in order to further reduce the number of instrumented collectives. Furthermore, our



(a) Compilation time overhead (b) Execution time overhead for HERA

Fig. 3: Overhead of average compilation time with and without verification code generation and execution time overhead for HERA with strong scaling



(a) Execution time overhead for NASPB (b) Execution time overhead for EulerMHD

Fig. 4: Execution time overhead for NASPB class C with strong scaling and for EulerMHD with weak scaling

approach in PARCOACH is a preliminary work setting the basis for a wider set of analysis combining static and dynamic aspects and extended to OpenMP and hybrid (OpenMP + MPI) parallelisms.

Acknowledgments

This work is (integrated and) supported by the PERFCLOUD project. A French FSN (Fond pour la Société Numérique) cooperative project that associates academics and industrials partners in order to design then provide building blocks for a new generation of HPC datacenters.

References

[1] GCC 4.7. `gcc.gnu.org/gcc-4.7/`.

[2] IMB. software.intel.com/en-us/articles/intel-mpi-benchmarksl.

[3] NASPB. http://www.nas.nasa.gov/software/NPB.

[4] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, S. Kumar, E. Lusk, R. Thakur, and J. L. Träff. MPI on a million processors. pages 20–30. EuroPVM/MPI, 2009.

[5] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. Efficiently computing static single assignment form and the control dependence graph. In *ACM TOPLAS*, pages 13(4):451–490, 1991.

[6] J. DeSouza, B. Kuhn, B. R. de Supinski, V. Samofalov, S. Zheltov, and S. Bratanov. Automated, scalable debugging of MPI programs with Intel Message Checker. In *SE-HPCS*, pages 78–82, 2005.

[7] C. Falzone, A. Chan, E. Lusk, and W. Gropp. Collective error detection for MPI collective operations. pages 138–147. PVM/MPI, 2005.

[8] C. Falzone, A. Chan, E. Lusk, and W. Gropp. A portable method for finding user errors in the usage of MPI collective operations. *IJHPCA*, 2007.

[9] T. Hilbrich, F. Hänsel, M. Schulz, B. R. de Supinski, M. S. Müller, and W. E. Nagel. Runtime MPI collective checking with tree-based overlay networks. pages 117–122. EuroMPI, 2013.

[10] T. Hilbrich, J. Protze, M. Schulz, B. R. de Supinski, and M. S. Müller. MPI runtime error detection with MUST: advances in deadlock detection. In *Supercomputing*, pages 30:1–30:11, 2012.

[11] T. Hilbrich, M. Schulz, B. de Supinski, and M. Müller. MUST: A scalable approach to runtime error detection in MPI programs. Parallel Tools Workshop, 2010.

[12] H. Jourdren. HERA: A hydrodynamic amr platform for multi-physics simulations. In J. Foster, E. Lutton, J. Miller, C. Ryan, and A. Tettamanzi, editors, *Adaptive Mesh Refinement - Theory and Application*, pages 283–294. LNCSE, 2005.

[13] B. Krammer, K. Bidmon, M. S. Müller, and M. M. Resch. MARMOT: An MPI analysis and checking tool. In *PARCO*, pages 493–500, 2003.

[14] G. Luecke, H. Chen, J. Coyle, J. Hoekstra, M. Kraeva, and Y. Zou. MPI-CHECK: a tool for checking Fortran 90 MPI programs. Concurrency and Computation: Practice and Experience, pages 15:93–100, 2003.

[15] J. Merrill. GENERIC and GIMPLE: A new tree representation for entire functions. GCC summit, 2003.

[16] E. Saillard, P. Carribault, and D. Barthou. Combining static and dynamic validation of MPI collective communications. pages 117–122. EuroMPI, 2013.

[17] S. Sharma, G. Gopalakrishnan, and R. M. Kirby. A survey of MPI related debuggers and tools. 2007.

[18] S. Siegel and T. Zirkel. Automatic formal verification of MPI based parallel programs. In *PPoPP*, pages 309–310, 2011.

[19] S. F. Siegel. Verifying Parallel Programs with MPI-Spin. In F. Cappello, T. Hrault, and J. Dongarra, editors, *PVM/MPI*, volume 4757 of *Lecture Notes in Computer Science*, pages 13–14. Springer, 2007.

[20] J. L. Träff and J. Worringen. Verifying collective MPI calls. pages 18–27. PVM/MPI, 2004.

[21] J. S. Vetter and B. R. de Supinski. Dynamic software testing of MPI applications with Umpire. In *Supercomputing*, pages 51–51. ACM/IEEE, 2000.

[22] 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 *Supercomputing*, pages 1–10, 2010.

[23] M. Wolff, S. Jaouen, and H. Jourdren. Hight-order dimensionally split lagrange-remap schemes for ideal magnetohydrodynamics. In *Discrete and Continuous Dynamical Systems Series S*. NMCF, 2009.