PARCOACH Extension for a Full-Interprocedural Collectives Verification

Pierre Huchant, Emmanuelle Saillard, Denis Barthou, Hugo Brunie, Patrick Carribault

To cite this version:

Pierre Huchant, Emmanuelle Saillard, Denis Barthou, Hugo Brunie, Patrick Carribault. PARCOACH Extension for a Full-Interprocedural Collectives Verification. Second International Workshop on Software Correctness for HPC Applications, Nov 2018, Dallas, United States. hal-01937316

HAL Id: hal-01937316
https://hal.inria.fr/hal-01937316

Submitted on 28 Nov 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

PARCOACH Extension for a Full-Interprocedural Collectives Verification

Pierre Huchant *†, Emmanuelle Saillard †, Denis Barthou *†, Hugo Brunie *†‡ and Patrick Carribault ‡
* U. of Bordeaux, Bordeaux INP — Bordeaux, France — name.lastname@inria.fr
† Inria Bordeaux Sud-Ouest — Bordeaux, France — name.lastname@inria.fr
‡ *CEA, DAM, DIF,* — F-91297 Arpajon, France — name.lastname@cea.fr

***Abstract*—The advent to exascale requires more scalable and efficient techniques to help developers to locate, analyze and correct errors in parallel applications. PARallel COntrol flow Anomaly CHecker (PARCOACH) is a framework that detects the origin of collective errors in applications using MPI and/or OpenMP. In MPI, such errors include collective operations mismatches. In OpenMP, a collective error can be a barrier not called by all tasks in a team. In this paper, we present an extension of PARCOACH which improves its collective errors detection. We show our analysis is more precise and accurate than the previous one on different benchmarks and real applications.**

***Keywords*-MPI, OpenMP, Collectives, Static analysis, Verification**

I. Introduction

One major type of error that can arise in parallel programs is deadlocks. A deadlock occurs in a parallel program if there are parallel tasks waiting for an event that is never going to happen. When a deadlock occurs in a parallel program, it is usually hard to identify what caused it. Some tools have been developed to help programmers in the debugging process but they often come with restrictions. In this paper, we focus on the detection of deadlocks in applications using MPI or OpenMP, the two most-used parallel programming models.

The Message Passing Interface [1] (MPI) is a distributed memory model where each MPI process executes a parallel instance of a program. Processes can exchange data mainly with the help of point-to-point and collective operations. These latter are always performed on a group or groups of processes (called communicator). Unlike MPI, OpenMP [2] is based on shared memory paradigm. An OpenMP program starts with a single thread of execution (master thread) which creates a team when it encounters a `#pragma omp parallel` construct. At the end of a parallel region, all threads of the team except the initial thread are asleep. The OpenMP standard offers many possibilities inside a parallel region. When needed, threads synchronize with a `barrier` that can be either explicit[1] or implicit[2]. Both models have what we call *collectives*. We define a *collective* as:

- Any blocking or non-blocking communication involving all MPI processes of a same communicator in MPI: `MPI_Barrier`, `MPI_Ibarrier`, `MPI_Bcast`, `MPI_Ibcast`, `MPI_Allreduce`, ...
- A barrier and any worksharing construct in OpenMP: `#pragma omp {barrier/single/for/sections/workshare}`.
  Note that even a worksharing construct with a `nowait` *clause is considered as a collective.*

MPI and OpenMP specifications share a common restriction about these collectives: all MPI processes / OpenMP threads must have the same sequence of collectives. This means that they must all call the same collectives, in the same order. A violation of this constraint can result in a deadlock or an unspecified behavior of the program. In the rest of the paper, we use *collective error* to refer to a misuse of a collective (i.e., collective mismatch or collective not called by all processes/threads). As an example, an OpenMP `single` region nested in another `single` is a collective error. In this context, a program is said correct if it has no collective error. Note that if a program is proved statically correct it is dynamically correct. The reverse is not true.

In this paper, we propose an extension of PARCOACH to detect collective errors in MPI and OpenMP applications and pinpoint their root causes. Our method can also be used to detect collectives in MPI+OpenMP applications. In that case, MPI and OpenMP collectives are checked in separate analyses. More precisely, we make the following contributions:

- New interprocedural analysis in PARCOACH
- Full integration into the LLVM compiler
- Comparison between the previous analysis and the new one on several MPI and OpenMP benchmarks and applications

A. PARCOACH

The PARallel COntrol flow Anomaly CHecker [3]–[6] (PARCOACH) is a framework that detects misuse of collectives in two steps. First, an intraprocedural static analysis studies the control flow of each function of a program to find *statically incorrect functions*: functions containing potential deadlocks [3]. During this step, warnings are issued with all conditionals potentially responsible for a deadlock. Then, all collectives inside *statically incorrect functions* are instrumented in order to verify the potential deadlocks at execution

[1] `#pragma omp barrier`

[2] There is an implicit barrier at the end of a parallel region as well as at the end of a worksharing region, unless a `nowait` clause is specified

time. Check functions are inserted before all collectives and return statement of the function. In case of an actual deadlock situation at runtime, the execution is stopped, displaying an error message with compilation information. PARCOACH aims at pinpointing the cause of collective errors and giving the more precise feedback to developers. In the rest of the paper, we use *intraprocedural* to refer to this analysis.

In [5], we propose a light improvement of PARCOACH static analysis to handle interprocedural information. The method keeps and reuses summaries of functions. Each function is replaced by the valid sequence of MPI collectives it contains (collective calls not depending on the control flow). However, this method is limited when there is an invalid sequence of collectives in the function. Similarly, we expose in [4] the same idea for OpenMP programs. Instead of keeping a valid sequence of collectives, we keep the minimal number of collectives contained in a function. For purpose of clarity, we use the term *summary-based interprocedural* analysis when referring to this method. This paper suggests a new interprocedural analysis that builds a parallel program control-flow graph to capture the control-flow of the whole program. Our method is more precise and accurate. We adapt the dynamic analysis to the new compile-time information and show the impact on execution-time.

B. Outline

Section II presents motivating examples of our work. Sections III and IV respectively describe the new static and dynamic analyses of PARCOACH. Section V shows experimental results. Finally, section VI gives an overview of related work about collective errors detection in MPI and OpenMP applications and section VII concludes the paper.

II. Motivating Examples

Figure 1 illustrates MPI and OpenMP examples. All codes have two functions: `f` and `c`, collectives are written in bold. For the MPI code 1, PARCOACH intraprocedural finds a potential deadlock in function `f` because of the conditional line 7 and issues a warning indicating the barrier line 8 may not be called by all processes. Indeed the intraprocedural analysis checks each function separately. However, there is a potential collective mismatch as the `else` statement calls function `c` that contains a non-blocking barrier (line 2). With the summary-based interprocedural analysis, `c` is replaced by `MPI_Ibarrier` and a warning reports the collective mismatch.

In MPI code 2, the intraprocedural analysis detects a potential deadlock in function `f` (collective sequences are `MPI_Barrier MPI_Barrier` in `then` and `MPI_Barrier MPI_Bcast MPI_Barrier` in `else`). This warning is removed with the summary-based interprocedural analysis when `c` is replaced by its summary ($\{MPI_Bcast\}$ is the valid sequence of collectives of function `c`).

In MPI code 3, PARCOACH intraprocedural will report an error for `MPI_Reduce` line 4 in `c` and `MPI_Barrier` line 9 in `f`. With the summary-based interprocedural analysis, a potential error is detected for `MPI_Reduce` only because of the conditional line 3 in `c`. Our new interprocedural analysis returns conditionals lines 3 and 8 as potentially leading to a collective error.

The same scenario is presented in MPI code 4. PARCOACH intraprocedural identifies the conditional line 2 in `c` as potentially leading to a deadlock, but not the conditional line 7 in `f`. The summary-based interprocedural analysis finds the same collective error and won't report any problem in `f` either. Indeed as there is no valid sequence of collectives, the summary of `c` is empty. And yet, the conditional line 7 is also responsible for a potential deadlock. Besides, if all processes eventually call the barrier in `c` and don't have the same value for the conditional in `f`, the feedback reported by the summary-based interprocedural analysis will be wrong. The new interprocedural analysis is exhaustive and pinpoints both conditionals lines 2 and 7 as potentially leading to a collective error. The same analysis can be applied to the OpenMP code 3 (same code written in OpenMP).

In MPI code 5, neither the intraprocedural analysis nor the summary-based interprocedural analysis detect a potential error. As our new analysis takes MPI communicators into account, a warning is emitted for barriers lines 7 and 9.

In the OpenMP code 1, the `single` line 2 may not be called by all OpenMP threads because of the conditional line 11. By analyzing `c` and `f` separately, PARCOACH intraprocedural doesn't detect any collective error. By replacing `c` with $n_{single} = 1$, the minimal number of the worksharing construct `single` in `c`, the summary-based interprocedural analysis identifies the conditional line 11 as the cause of a possible deadlock.

In the OpenMP code 2, the two `section` regions contain a call to `c` which contains a barrier. By default, these two regions will be executed once by two different threads. As there is an implicit barrier at the end of the `sections` construct, all threads will synchronize through distinct barriers. This is not detected by the intraprocedural analysis and may lead to a deadlock situation.

When a function is statically not verifiable, the summary of the function kept by the summary-based interprocedural analysis is incomplete. This prevents from reporting correct and precise feedback as the analysis can miss the real cause of a deadlock.

III. Full-interprocedural Analysis

PARCOACH static analysis takes place in the middle of the compilation chain, where each function of a program is represented by a Control Flow Graph (CFG). In a CFG, a node can be either a basic block or the entry/exit point of a function and edges represent possible flow of control between nodes. For the needs of PARCOACH analysis, all CFGs are augmented with collective information: all nodes containing collectives (*collective nodes*) are tagged. For OpenMP

```
void c(){
 MPI_Ibarrier(com,..);
 ...
}

void f(){
 if(..)
  MPI_Barrier(com);
 else
  c();
}
```

(a) MPI Code 1

```
void c() {
  MPI_Bcast(..);
}

void f() {
  if (..) {
    MPI_Barrier(com);
    c();
    MPI_Barrier(com);
  } else {
    MPI_Barrier(com);
    MPI_Bcast(..);
    MPI_Barrier(com);
  }
}
```

(b) MPI Code 2

```
void c(){
 MPI_Barrier(com);
 if(..)
   MPI_Reduce(..);
}

void f(){
 if(..)
  MPI_Barrier(com);
 else
  c();
}
```

(c) MPI Code 3

```
void c(){
 if(..)
  MPI_Barrier(com);
}

void f(){
 if(..)
  c();
}
```

(d) MPI Code 4

```
void c(){
  MPI_Reduce(..);
}

void f(){
 if(..)
  MPI_Barrier(com1);
 else
  MPI_Barrier(com2);
 ...
 c();
}
```

(e) MPI Code 5

```
void c(){
 #pragma omp single
 {
   /* ... */
 }
}

void f(){
 #pragma omp parallel
 {
   if(..)
     c();
 }
}
```

(f) OpenMP Code 1

```
void c(){
  /* ... */
  #pragma omp barrier;
}

void f(){
 #pragma omp parallel
 {
   #pragma omp sections
   {
     #pragma omp section
     {
        c();
     }
     #pragma omp section
     {
        c();
     }
   }
 }
}
```

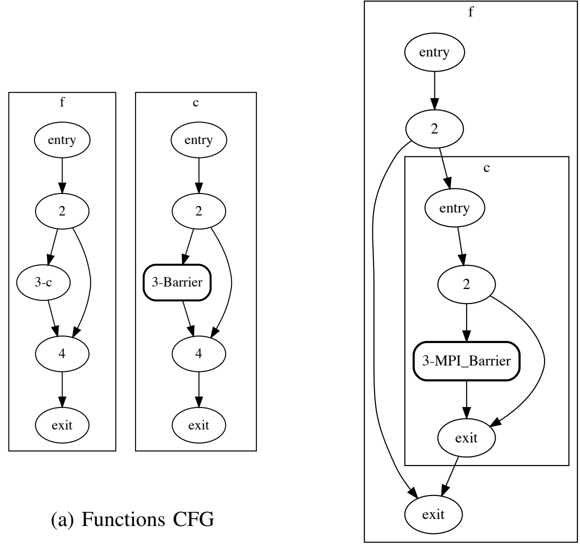
(g) OpenMP Code 2

```
void c(){
 if(..)
   #pragma omp barrier
}

void f(){
 #pragma omp parallel
 {
   if(..)
     c();
 }
}
```

(h) OpenMP Code 3

Fig. 1: Examples of MPI and OpenMP codes.



(a) Functions CFG

(b) PPCFG

Fig. 2: MPI Code 4 functions CFG (left) and the corresponding PPCFG (right)

programs, PARCOACH uses the OMPCFG representation described in [7]. The OMPCFG modifies the CFG by creating new nodes to isolate OpenMP directives and adding edges between previous nodes and the new ones according to the OpenMP semantics. Hence, nodes containing `master`, `for` and `single` directives are considered as conditionals; `sections` and `workshare` are considered as switch.

In the following, we use the notion of *iterated postdominance frontier* (PDF^+). The iterated postdominance frontier of a node n corresponds to the control-flow divergences that may result in the execution or non-execution of n.

The next two sections describe our new interprocedural analysis, referred as *full-interprocedural* analysis.

A. PPCFG Construction

The full-interprocedural analysis builds a parallel program control flow graph (PPCFG) in order to get interprocedural information. We extend the intermediate representation used by PARCOACH by replacing each callsite by its CFG. In order to reduce the cost of the interprocedural analysis, each function CFG is first reduced. Only nodes with collectives, those inside the PDF^+ of these nodes, function entry and exit nodes are kept. All other nodes are removed. The edges among the nodes keep the relation of successor and predecessor existing in the initial CFG.

Figure 2b illustrates the PPCFG of the example presented Figure 1c. The PPCFG is built based on the initial functions CFG presented Figure 2a. Thick nodes are collective nodes, boxes represent functions.

B. Collective Error Detection

Our analysis studies the PPCFG to find nodes conducting to paths with different sequences of collectives (i.e., not the same number or not the same collectives). With a graph traversal of the PPCFG, we compute the possible *execution order* (i.e.,

calling order) of each collective and the PDF^+ for collectives of the same type and order. Nodes in the PDF^+ represent all conditionals possibly responsible for a deadlock. In the MPI code 4, the barrier has an execution order 0 (first collective encountered). Conditionals lines 2 and 7 are in the PDF^+ of the instruction corresponding to `MPI_Barrier`.

Algorithm 1 describes the collective errors detection for MPI and OpenMP programs. The algorithm takes as input the $PPCFG$ of a program and returns the set O containing the information needed to give a precise feedback to users (collective name, line of collectives and conditionals in the source code) and the set of all conditionals potentially responsible for a deadlock. For MPI applications, PARCOACH analyzes the program separately for each communicator.

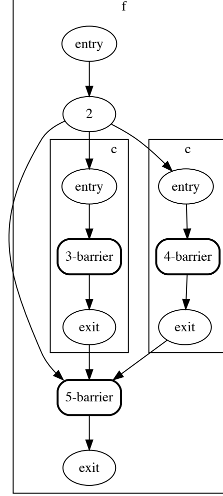
Note that we use the verbosity level 0, as defined in [4] (relaxed verification) for OpenMP collectives verification.

Data: $PPCFG$
$O \leftarrow \emptyset$ ▷ Output set
Remove loop backedges in $PPCFG = (V, E)$ to compute execution orders for each collective
for r *in node orders* **do**
  for c *in collective names of execution order* r **do**
    $C_{r,c} \leftarrow \{u \in V | r$ is the max. execution order of u, u executes a collective with name $c\}$
    if $PDF^+(C_{r,c}) \neq \emptyset$ **then**
      $O \leftarrow O \cup (c, PDF^+(C_{r,c}))$
  end
  for *each collective* c *in a loop* **do**
    $O \leftarrow O \cup (c, \{$ loop exit nodes $\})$
  end
end
Output nodes in O as warnings
return O

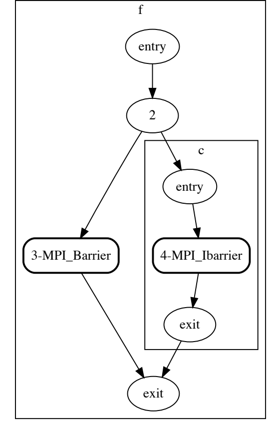
Algorithm 1: Full-interprocedural Control-flow Analysis

Figure 3 shows PPCFGs of examples figure 1g and 1a. The algorithm first computes execution order of collectives. In figure 3a, barriers in nodes 3 and 4 can be the first collectives encountered (paths $entry_f \rightarrow 2 \rightarrow entry_c \rightarrow 3$ *or* 4). The barrier node 5 can be the first collective (paths $entry_f \rightarrow 2 \rightarrow 5$) or the second collective encountered. The algorithm then considers $C_{0,barrier} = \{3, 4\}$ and $C_{1,barrier} = \{5\}$ with $PDF^+(C_{0,barrier}) = \{2\}$ and $PDF^+(C_{1,barrier}) = \emptyset$. Because $PDF^+(C_{0,barrier}) \neq \emptyset$, a warning will be issued for the conditional node 2. The PPCFG figure 3b has two collectives: `MPI_Barrier` node 3 and `MPI_Ibarrier` node 4. The static analysis detects a potential collective error for the barrier in 3 and the non-blocking barrier in 4 and reports a warning for the conditional located in 2 ($PDF^+(C_{0,Barrier}) = \{2\}$ and $PDF^+(C_{0,Ibarrier}) = \{2\}$).

Because potential errors found at compilation-time may not be correlated with actual control-flow (false positive), the next section presents a static instrumentation of the code to check if all potential deadlocks will eventually occur during execution.



(a) OpenMP code 2 PPCFG



(b) MPI Code 1 PPCFG

Fig. 3: MPI Code 1 and OpenMP Code 2 PPCFG

IV. Code Instrumentation

With the help of the static analysis, we perform a selective instrumentation of programs. Only statically not verifiable programs are instrumented (no warning is issued during compilation time for statically correct programs so no instrumentation is done).

For statically not verifiable programs, all collectives and exit statements are instrumented with Check Collective (`CC`) functions. The instrumentation starts from the first collectives that may deadlock in the program. Relying on the work in [3], we define the `CC` function as follows. CC functions take as input an integer i_{model} identifying the parallel programming model used, the communicator related to the collective (0 in case of OpenMP), an integer i_c identifying the collective and the set O generated at compile-time. Through `CC` calls, processes/threads can verify which collectives will be called at different steps of execution. For MPI programs, `CC` calls a `MPI_Reduce` with a new MPI operator returning -1 if there is at least two different integers i_c among processes. For OpenMP programs, each thread updates a shared variable relied to i_c. When a deadlock is about to occur, an error message is returned with compilation information (related warnings). Figure 4 shows an example of MPI code instrumented (inspired from MPI Code 4 figure 1d). A `CC` function is inserted before the collective `MPI_Barrier` in `c` and `MPI_Finalize` in `main`. The `MPI_Barrier` line 8 is called by all processes and therefore not instrumented.

We denote the sequence of collective calls executed by a process/thread in a program execution as $c_1 c_2 ... c_n$ with c_i the i-th collective called. Our instrumentation rewrites each collective c_j into $s_j c_j$ corresponding to the `CC` function with integer j and the initial collective c_j, starting with collectives that may deadlock. A CC function s_0 is added after all collectives

```
1  void c() {
2    if(..)
3      MPI_Barrier(com);
4  }
5
6  int main() {
7    /* .. */
8    MPI_Barrier(com);
9
10   if (..)
11     c();
12
13   MPI_Finalize();
14 }
```

(a) MPI Code

```
1  void c() {
2    if(..) {
3      CC(MPI, com, i_barrier, O);
4      MPI_Barrier(com);
5    }
6  }
7
8  int main() {
9    /* .. */
10   MPI_Barrier(com);
11
12   if (..)
13     c();
14
15   CC(MPI, com, 0, ∅);
16   MPI_Finalize();
17 }
```

(b) MPI Code instrumented

Fig. 4: MPI example and its instrumentation.

(CC before exit/abort/MPI_Finalize). Assuming the first collective that may deadlock is the k-th collective, a sequence $c_1c_2...c_{k-1}c_k...c_n$ then becomes $c_1c_2...c_{k-1}s_kc_k...s_nc_ns_0$. If all collectives sequences are the same for all processes/threads, no instrumentation is done and the collectives sequences are still identical. The instrumentation does not introduce deadlocks. If a program deadlocks due to collective operations, we have the two following scenarios:

- A process/thread calls a collective c_i while another process/thread calls a collective c_j with $i \neq j$. The collectives sequences of both processes/threads only differ with their last collective and are prefixed by $c_1...c_{i-1}$. The instrumentation changes both collectives sequences into $c_1...c_{i-1}s_i$ and $c_1...c_{i-1}s_j$. The sequences stop with s_i and s_j since $CC(-,-,i,-)$ and $CC(-,-,j,-)$ lead to an error detection and abort. The modified program no longer deadlocks.
- A process/thread calls a collective while another one exits the program. The collectives sequence of the process/thread exiting the program is $c_1...c_{i-1}$ and the process/thread calling the collective executes the same prefix sequence with one more collective c_i. The instrumentation changes both collectives sequences into $c_1...c_{i-1}s_0$ and $c_1...c_{i-1}s_i$. The sequences stop with s_0 and s_i since $CC(-,-,0,-)$ and $CC(-,-,i,-)$ lead to an error detection and abort. Again, the modified program does not deadlock.
- A process/thread calls a collective while another one calls a blocking operation (e.g., point to point operation in MPI). This case is not supported by our analysis.

V. Experimental Results

PARCOACH was previously implemented as a GCC plugin, working with GCC version 4.7.0. The summary-based interprocedural analysis was implemented as a python script working on GCC dumped traces. We integrated both the intraprocedural and full-interprocedural analyses into the LLVM [8] compiler framework, version 3.9. PARCOACH[3] is implemented as an open source LLVM pass.

Our new static analysis is done at the LLVM IR level and applies algorithm 1 and the code instrumentation. It uses several LLVM existing pass to get loops and dominance/-postdominance information. PARCOACH is independent from MPI implementation and handles all blocking and nonblocking collectives. For runtime checking, the application needs to be linked to our dynamic library (which contains CC function implementation).

There exists no benchmark or application with deadlock. To evaluate the efficiency of our tool, we manually introduced errors in small codes. PARCOACH was always able to detect them. In this section, we present results we obtained on C programs: MILC [9], Gadget-2 [10] and MPI-PHYLIP [11] applications, AMG [12] from the CORAL benchmarks, the High-Performance Linpack benchmark [13] (HPL), miniAMR and CoMD from the Mantevo project [14], IOR [15] from the NERSC benchmarks (IOR-POSIX and IOR-MPIIO), Hydro [16], and IS from the NAS benchmarks [17]. These benchmarks have been chosen because they contain collectives and are written in C, also they cover a wide spectrum of HPC scientifc domains.

Table I shows benchmarks and applications statistics. The second column depicts the parallel programming model used. The third and fourth columns respectively give the number of functions and collectives found in programs. The last column gives the number of communicators for MPI applications. MILC and MPI-PHYLIP are represented by the cumulative sum of all mini applications they contain. AMG is parallelized with MPI and OpenMP.

TABLE I: Applications and Benchmarks Statistics.

Application	Parallelism	# func.	# coll.	# com.
MILC*	MPI	24,242	635	253
Gadget-2	MPI	193	70	1
MPI-PHYLIP*	MPI	4,000	128	12
Bench. / mini app.	**Parallelism**	**# func.**	**# coll.**	**# com.**
Coral AMG	MPI	1,207	79	19
	OpenMP	1,207	11	-
HPL	MPI	193	3	1
miniAMR	MPI	103	43	2
IOR-POSIX	MPI	175	82	5
IOR-MPIIO	MPI	197	88	5
Hydro	MPI	99	13	1
CoMD	MPI	124	8	1
NAS-MPI IS	MPI	36	9	1
NAS-OMP IS	OpenMP	51	3	-

A. Static Analysis Results

Table II depicts the number of warnings and conditionals returned by both intraprocedural and full-interprocedural analyses for all benchmarks and applications. We can notice that NAS-OMP IS is collective error free as no warning is emitted at compile time for this benchmark.

A more detailed ratio is presented figures 5 and 6. Figure 6 gives the number of conditionals added and removed with PARCOACH using the full-interprocedural method compared

[3] PARCOACH is available at https://esaillar.github.io/PARCOACH/

TABLE II: Number of warnings reported and conditionals responsible for a collective error for both intraprocedural and full-interprocedural analyses

Application	**Intraprocedural**		**full-interprocedural**	
	#warn.	**#cond.**	**#warn.**	**#cond.**
MILC*	114	114	498	2195
Gadget-2	21	22	68	30
MPI-PHYLIP*	65	44	65	44
Bench. / mini app.	**Intraprocedural**		**full-interprocedural**	
	#warn.	**#cond.**	**#warn.**	**#cond.**
Coral AMG	45	34	76	169
	6	2	11	48
HPL	2	1	2	1
miniAMR	20	15	32	36
IOR-POSIX	67	64	82	79
IOR-MPIIO	73	68	88	83
Hydro	11	11	13	12
CoMD	0	0	8	3
NAS-MPI IS	3	1	3	1
NAS-OMP IS	0	0	0	0

to the intraprocedural analysis while figure 5 gives the number of warnings added and removed. Warnings reported by the full-interprocedural analysis are mostly new warnings and few warnings were removed. The number of conditionals added and removed is also unbalanced. Adding (resp. removing) a conditional does not necessary imply adding (resp. removing) a warning since two conditionals can be responsible for the same warning.

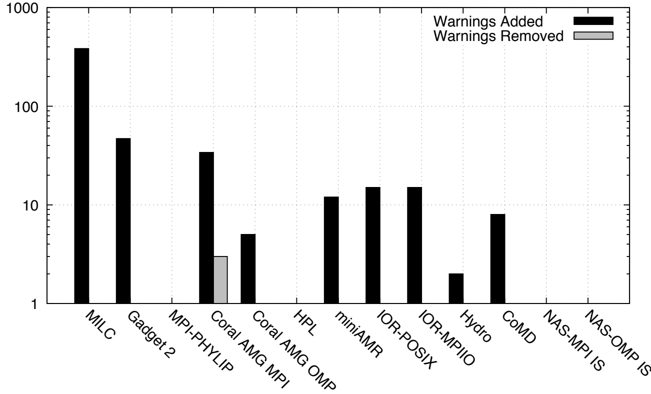


Fig. 5: Number of warnings added and removed with PARCOACH using the full-interprocedural method compared to PARCOACH using the intraprocedural analysis

Figure 7 shows the overhead induced when using the full-interprocedural analysis in PARCOACH. The compilation time can be around third time the initial time (Coral AMG OMP). However, as the corresponding total compilation time with our analysis is 1 minute, we think it is acceptable.

When reporting a potential collective error, PARCOACH pinpoints the source of the error. As an example, it reports the following warning for the MPI code figure 1a:

```
PARCOACH: warning: MPI_Ibarrier line 2
possibly not called by all processes
```

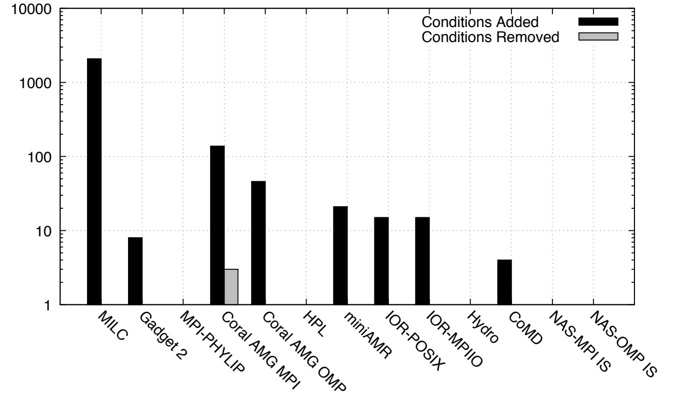


Fig. 6: Number of conditionals added and removed with PARCOACH using the full-interprocedural method compared to PARCOACH using the intraprocedural analysis

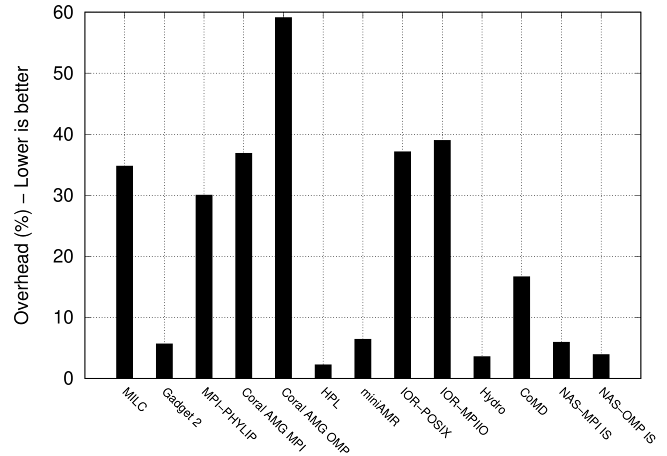


Fig. 7: Compile-time overhead using the full-interprocedural analysis (ratio between the PARCOACH analysis time and the total compilation time).

```
because of conditional(s) line(s) 7
PARCOACH: warning: MPI_Barrier line 8
possibly not called by all processes
because of conditional(s) line(s) 7
```

B. Execution Results

In order to realize the usability of our tool, we tested our code instrumentation on the Hydro benchmark. Hydro solves compressible Euler equations of hydrodynamics. We use the fine grain MPI version using C of the benchmark. Results were obtained on the Cori (Cray-XC40) supercomputer, deployed at NERSC [18] and averaged (over 50 runs for Hydro). Cori is composed of two partitions. One has $2,388$ Intel Xeon "Haswell" nodes with 32 cores each and the other contains $9,688$ Intel Xeon Phi (KNL) nodes. In this section, *Reference* denotes the original version of a benchmark.

Figure 8 shows the execution-time of Hydro for a range of MPI processes from 32 to 320. As can be seen in the figure, the overhead induced by PARCOACH runtime verification is

under 6%.

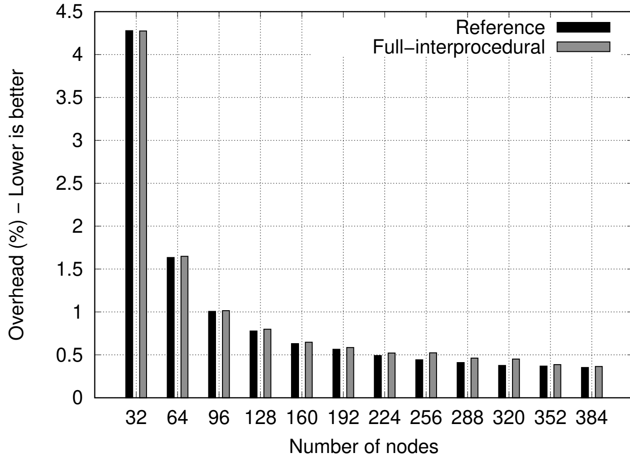


Fig. 8: Execution-Time of Hydro with and without runtime verification (domain size = 500x500, nstepmax=200)

When a deadlock is about to occur, PARCOACH stops the execution and reports an error message with compilation information. For the code presented figure 1a, PARCOACH reports the following error message:

```
PARCOACH: Error detected on rank 0
Abort is invoking line 8 before calling
MPI_Barrier in MPIcode1.c
See warning(s): MPI_Barrier line 8
possibly not called by all processes
because of conditional(s) line(s) 7,
MPI_Ibarrier line 2  possibly not called
by all processes because of
conditional(s) line(s) 7
```

This feedback helps fixing the deadlock. For example the non-blocking barrier can be replaced by a blocking one.

VI. Related Work

This section summarizes existing tools for collective error detection in MPI and OpenMP programs.

a) MPI: MPI collective communications are crucial for many large-scale applications. That is why it is relevant to assist developers debugging these operations. Although collective errors detection is mainly done at execution-time, some static tools have emerged. These tools have the advantage of not requiring the execution of applications and are input data set independent but can produce false positives. Among static tools, we can mention MPI-SPIN and its successor TASS [19] that use model checking and symbolic verification. These tools face a combinatorial number of program states.

Although dynamic tools are input data set dependent and can only detect an error when it is about to occur, they better manage huge number of processes compared to static analyses. Some dynamic tools detect deadlocks with a time-out approach, which may cause false positive. It is the case of the Intel Trace Analyzer and Collector [20] (ITAC) and DAMPI [21]. Although PARCOACH static analysis may cause false positives, the program instrumentation assures only real deadlocks are catched at runtime. MUST [22], [23] is able to check MPI collective operations with an offloading approach using wait-for graphs. Compared to MUST, PARCOACH stops the execution before a deadlock occurs and gives a more precise feedback about what caused it. STAT [24] uses a post-mortem analysis. It studies the stack trace of execution to detect deadlock situations. This method does not allow to find deadlock root causes.

An extension of MPICH directly verifies collective operations inside the MPI implementation [25]. This method is therefore limited to the information available in the MPI routines.

PARCOACH [3], [5] was designed to take the best of static and dynamic methods. It combines a scalable static analysis based on the study of programs control flow with an instrumentation of the code that verifies potential deadlocks at execution-time. In case of a deadlock situation, PARCOACH stops the program before the deadlock occurs and returns feedback about what caused it. PARCOACH only checks if the sequence of collectives is deterministic and supposes all MPI blocking and nonblocking collectives are called with compatible arguments. In [5], we compute summaries of functions in addition to the original static analysis. As shown in section II with MPI codes 4 and 5, even if this solution can improve the intraprocedural static analysis, it reports incomplete root causes of deadlocks. We have extended PARCOACH with a full-interprocedural analysis that corrects the previous PARCOACH analyses and pinpoints all sources of collective errors in programs.

b) OpenMP: Deadlock detection in OpenMP programs is either static or dynamic.

The OpenMP Analysis Toolkit (OAT) [26] relies on symbolic analysis to detect concurrency errors, including deadlocks. It encodes OpenMP regions into SMT formulas and uses the SMT-solver Yices to detect errors. Zhang *et al.* [27] detect textually unaligned barriers with an interprocedural concurrency analysis. This one uses the control flow of a program and a barrier tree. Our method is simpler: we build a parallel program control-flow graph to get interprocedural information. Compilers like GCC [28], ICC [29] or LLVM [8] issue either a warning or an error message for invalid nesting of regions. For example, GCC issues a warning for a `single` directive in another `single` directive whereas ICC and LLVM return an error message. However, if the nested region is encapsulated into a function, they don't detect anything. PARCOACH [4] uses the same static/dynamic method as for MPI programs to detect misuse of barriers and worksharing constructs in OpenMP programs.

Intel Thread checker [30], [31] (now superseded by Intel Inspector XE [32]) and Sun thread analyzer [31], [33] both use code instrumentation to collect operations on memory, thread management and synchronization at runtime. These

operations are recorded in a trace file which is then analyzed in order to find deadlocks. This post-mortem method has the same drawback as the dynamic methods, it only finds errors related to the parts of the program that have been executed and is dependent to the input data set.

VII. Conclusion

In this paper, we present an extension of the PARCOACH framework. PARCOACH detects misuse of collectives in MPI and OpenMP programs by combining static and dynamic analyses. Our extension uses a parallel program control-flow graph for a more precise and accurate interprocedural analysis. This analysis brings an acceptable overhead that is not far from the overhead induced by the previous PARCOACH analysis. Furthermore, we have shown that our runtime verification has a low overhead (less than 6%) on the Hydro benchmark.

The work we present in the paper is focused on MPI and OpenMP programs but can easily be adapted to any programming models with the same collectives constraint (e.g., CUDA or UPC). Furthermore, our method can be used to help developers to verify MPI and OpenMP collectives in their MPI+OpenMP applications. As a future work, we intent to couple our analysis with a data-flow analysis to reduce the number of false positives. We also intent to improve our code instrumentation. For example, we could instrument only portions of code that may deadlock in programs. The first collectives potentially deadlocking could be instrumented and all following ones until reset points.

References

[1] http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf, 2012, message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 3.0.

[2] http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf, 2015, openMP 4.5 Complete Specifications.

[3] E. Saillard, P. Carribault, and D. Barthou, "Parcoach: Combining static and dynamic validation of mpi collective communications," *The International Journal of High Performance Computing Applications*, vol. 28, no. 4, pp. 425–434, 2014.

[4] ——, "Static Validation of Barriers and Worksharing Constructs in OpenMP Applications," in *International Workshop on OpenMP*, Salvador, Brazil, Sep. 2014, pp. 73 – 86. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01078759

[5] E. Saillard, H. Brunie, P. Carribault, and D. Barthou, "PARCOACH Extension for Hybrid Applications with Interprocedural Analysis," in *9th International Workshop on Parallel Tools for High Performance Computing*, Dresden, Germany, Sep. 2015, pp. 135 – 146.

[6] J. Jaeger, E. Saillard, P. Carribault, and D. Barthou, "Correctness Analysis of MPI-3 Non-Blocking Communications in PARCOACH," in *EuroMPI*, 2015.

[7] Y. Lin, *Static Nonconcurrency Analysis of OpenMP Programs*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 36–50.

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

[9] "MILC." http://www.physics.utah.edu/~detar/milc/milc_qcd.html, 2016.

[10] V. Springel, "The cosmological simulation code gadget-2," *Monthly Notices of the Royal Astronomical Society*, vol. 364, pp. 1105–1134, 2005.

[11] A. J. Ropelewski, H. B. Nicholas, Jr, and R. R. Gonzalez Mendez, "MPI-PHYLIP: Parallelizing Computationally Intensive Phylogenetic Analysis Routines for the Analysis of Large Protein Families," *PLOS ONE*, vol. 5, no. 11, pp. 1–8, 11 2010.

[12] "CORAL AMG," https://asc.llnl.gov/CORAL-benchmarks/Summaries/AMG2013_Summary_v2.3.pdf, 2013.

[13] "High-Performance Linpack benchmark." http://www.netlib.org/benchmark/hpl/, 2016.

[14] 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, 2009.

[15] "NERSC IOR." http://www.nersc.gov/research-and-development/apex/apex-benchmarks/ior/, 2016.

[16] "Hydro Benchmark." https://github.com/HydroBench/Hydro, 2017.

[17] 2016, nASPB site: https://www.nas.nasa.gov/publications/npb.html.

[18] *Cori*, 2017, http://www.nersc.gov/users/computational-systems/cori/.

[19] S. F. Siegel and T. K. Zirkel, "Automatic Formal Verification of MPI-based Parallel Programs," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '11. New York, NY, USA: ACM, 2011, pp. 309–310.

[20] P. Ohly and W. Krotz-Vogel, "Automated MPI Correctness Checking: What if there was a magic option?" in *Proceedings of the 8th LCI International Conference on High-Performance Clustered Computing*, South Lake Tahoe, California, USA, 2007.

[21] A. Vo, S. Aananthakrishnan, G. Gopalakrishnan, B. R. d. de 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. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10.

[22] T. Hilbrich, B. R. de Supinski, F. Hänsel, M. S. Müller, M. Schulz, and W. E. Nagel, "Runtime MPI Collective Checking with Tree-based Overlay Networks," in *Proceedings of the 20th European MPI Users' Group Meeting*, ser. EuroMPI '13. New York, NY, USA: ACM, 2013, pp. 129–134.

[23] T. Hilbrich, B. R. de Supinski, M. Schulz, and M. S. Müller, "A Graph Based Approach for MPI Deadlock Detection," in *Proceedings of the 23rd International Conference on Supercomputing*, ser. ICS '09. New York, NY, USA: ACM, 2009, pp. 296–305. [Online]. Available: http://doi.acm.org/10.1145/1542275.1542319

[24] D. C. Arnold, D. H. Ahn, B. R. de Supinski, G. L. Lee, B. P. Miller, and M. Schulz, "Stack Trace Analysis for Large Scale Debugging," in *2007 IEEE International Parallel and Distributed Processing Symposium*, March 2007, pp. 1–10.

[25] J. L. Träff and J. Worringen, "Verifying collective MPI calls," in *PVM/MPI*. Springer, 2004, pp. 18–27.

[26] H. Ma, S. R. Diersen, L. Wang, C. Liao, D. Quinlan, and Z. Yang, "Symbolic Analysis of Concurrency Errors in OpenMP Programs," in *PARCO*, ser. ICPP, vol. 00, 2013, pp. 510–516.

[27] Y. Zhang, E. Duesterwald, and G. R. Gao, *Concurrency Analysis for Shared Memory Programs with Textually Unaligned Barriers*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 95–109.

[28] "The GNU Compiler Collection," https://gcc.gnu.org.

[29] "The Intel Compiler," https://software.intel.com/en-us/intel-compilers.

[30] U. Banerjee, B. Bliss, Z. Ma, and P. petersen, "Unraveling data race detection in the intel thread checker," in *Int'l. Symp. on Computer Architecture*, ser. ISCA, 2008.

[31] C. Terboven, "Comparing Intel Thread Checker and Sun Thread Analyzer," in *PARCO*, ser. Advances in Parallel Computing, C. H. Bischof, H. M. Bücker, P. Gibbon, G. R. Joubert, T. Lippert, B. Mohr, and F. J. Peters, Eds., vol. 15. IOS Press, 2007, pp. 669–676.

[32] "Intel Inspector XE," https://software.intel.com/en-us/intel-inspector-xe, 2017.

[33] "Thread Analyzer user guide," https://docs.oracle.com/cd/E19205-01/820-0619/index.html, 2010.