Multi-Valued Expression Analysis for Collective Checking

Pierre Huchant, Emmanuelle Saillard, Denis Barthou, Patrick Carribault

Multi-Valued Expression Analysis for Collective Checking

Pierre Huchant[1,2], Emmanuelle Saillard[2], Denis Barthou[1,2], and Patrick Carribault[3]

[1] Bordeaux Institute of Technology, U. of Bordeaux, LaBRI, Bordeaux, France
[2] Inria, Bordeaux, France
[3] CEA, DAM, DIF, F-91297 Arpajon, France

Abstract. Determining if a parallel program behaves as expected on any execution is challenging due to non-deterministic executions. Static analyses help to detect all execution paths that can be executed concurrently by identifying multi-valued expressions, i.e. expressions evaluated differently among processes. This can be used to find *collective errors* in parallel programs. In this paper, we propose a new method that combines a control-flow analysis with a multi-valued expressions detection to find such errors. We implemented our method in the PARCOACH framework and successfully analyzed parallel applications using MPI, OpenMP, UPC and CUDA.

1 Introduction

Collective operations and in particular synchronizations are widely used operations in parallel programs. They are part of languages for distributed parallelism such as MPI or PGAS (collective communications), shared-memory models like OpenMP (barriers) and languages for accelerators such as CUDA (synchronization within thread blocks, cooperative groups and at warp-level). A valid use of collective operations requires at least that their sequence is the same for all threads/processes during a parallel execution. An invalid use (*collective error*) leads to deadlocks or undefined memory state that may be difficult to reproduce and debug. Indeed, these languages do not require that all processes reach the same textual collective statement (*textual alignment* property [1, 2]). Finding which collective matches a given collective is needed for collective checking and requires to analyse the different concurrent execution paths of a parallel execution.

Aiken and Gay introduced the concept of *structural correctness* for synchronizations in SPMD programs, based on the notion of *multi-valued* and *single-valued* variables [3]. A variable is said multi-valued if its value is dependent on the process identifier (single-valued otherwise). A program has structurally correct synchronization if all processes have the same sequence of synchronization operations. Thus, if a synchronization is executed conditionally, both branches of the condition have a synchronization or the condition expression is single-valued.

We can extend the notion of structural correctness to collectives. In this paper, we propose a novel method to detect *collective errors* in parallel programs. It combines an inter-procedural analysis to perform collective matching and a data-flow analysis to detect multi-valued variables. The first pass finds control-flow divergence that may lead to collective deadlocks while the second one filters out the divergences that do not depend on process identifier. We show on several benchmarks and applications that this combination is more accurate than the state-of-the-art analyses and resolves some correctness issues. The analysis has been implemented in the PARallel COntrol flow Anomaly CHecker [4, 5] (PARCOACH) framework and tested on benchmarks and real HPC applications, using MPI, OpenMP, UPC and CUDA.

Section 2 describes several deadlock situations in parallel programs. Section 3 presents PARCOACH analysis while sections 4 and 5 describe our multi-valued expression detection and its integration in PARCOACH to find collective errors. Section 6 gives related work on dependence analyses and verification tools. Section 7 shows experimental results and Section 8 concludes our work.

2 Motivation

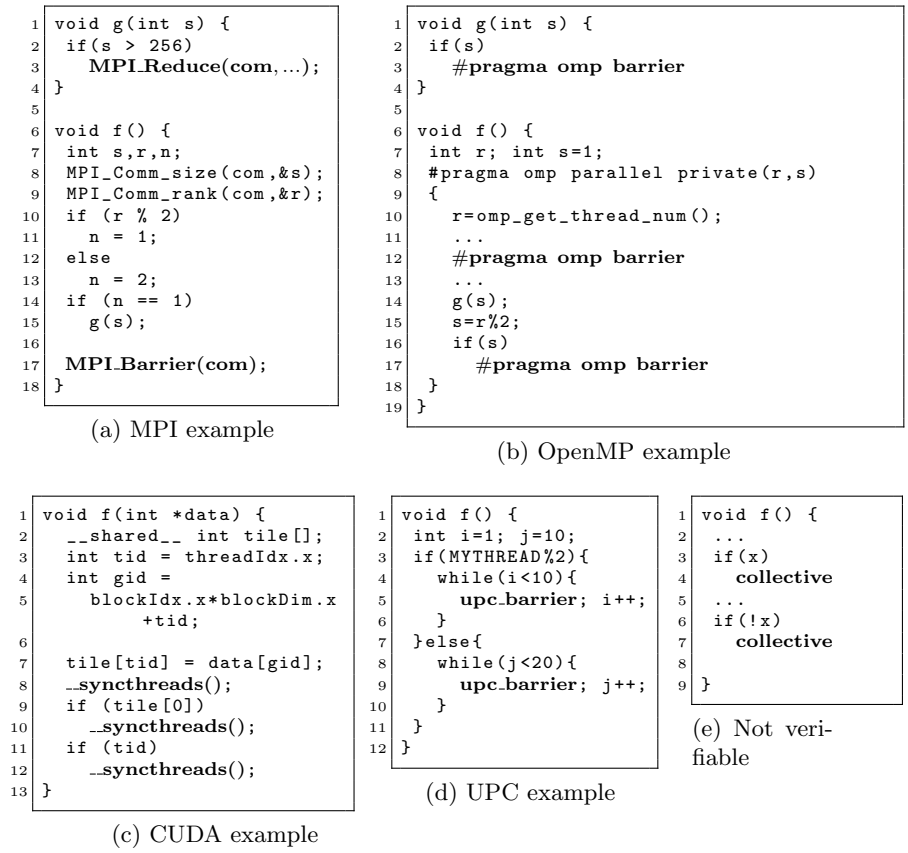
This section illustrates four possible deadlock situations due to collectives in MPI, OpenMP, CUDA and UPC as presented in Figure 1 (from a to d).

The MPI code (Fig.1a) contains two collectives: `MPI_Barrier` and `MPI_Reduce`. The call to `MPI_Barrier` at line 17 is performed by all MPI processes, whereas the call `MPI_Reduce` in `g` at line 3 may deadlock. Indeed, variable `n` is multi-valued in the condition expression line 14. Odd-ranked processes evaluate the conditional to true and potentially execute the reduce, while even-ranked ones evaluate it to false, hanging in the `MPI_Barrier` at line 17. On the contrary, variable `s` is single-valued. Hence, all processes in `g` execute the reduce or none. The goal of our method is to statically report this situation, identifying the conditional at line 14, and only this one, as a potential cause for mismatched calls.

According to the OpenMP specification, the same explicit and implicit[4] barriers (syntactically) should be executed by all threads. In practice, the OpenMP runtimes allow the execution of syntactically different barriers, as long as all threads execute the same number of barriers. The code Figure 1b is written in OpenMP. The `#pragma omp parallel` directive in function `f` defines `r`, containing the thread ID (l.10) and `s`, as private variables. The first barrier line 12 is encountered by all threads as it is in the global control flow. The barrier line 3 is either executed by all threads of the program or by none of them as `s` is single-valued when entering function `g` at line 14. Because `s` becomes multi-valued at line 15, the barrier line 17 is conditionally executed. This leads to a deadlock situation if the number of threads is greater than 1 at runtime.

The CUDA code (Fig.1c) manipulates multidimensional thread IDs through predefined variables such as `threadIdx`. In CUDA, synchronizations are valid if

[4] There is an implicit barrier at the end of all worksharing constructs, unless a `nowait` clause is specified.



```
1  void g(int s) {
2   if(s > 256)
3     MPI_Reduce(com,...);
4  }
5
6  void f() {
7   int s,r,n;
8   MPI_Comm_size(com,&s);
9   MPI_Comm_rank(com,&r);
10  if (r % 2)
11    n = 1;
12  else
13    n = 2;
14  if (n == 1)
15    g(s);
16
17  MPI_Barrier(com);
18 }
```

(a) MPI example

```
1  void g(int s) {
2   if(s)
3     #pragma omp barrier
4  }
5
6  void f() {
7   int r; int s=1;
8   #pragma omp parallel private(r,s)
9   {
10    r=omp_get_thread_num();
11    ...
12    #pragma omp barrier
13    ...
14    g(s);
15    s=r%2;
16    if(s)
17      #pragma omp barrier
18  }
19 }
```

(b) OpenMP example

```
1  void f(int *data) {
2    __shared__ int tile[];
3    int tid = threadIdx.x;
4    int gid =
5      blockIdx.x*blockDim.x
           +tid;
6
7    tile[tid] = data[gid];
8    __syncthreads();
9    if (tile[0])
10     __syncthreads();
11   if (tid)
12     __syncthreads();
13 }
```

(c) CUDA example

```
1  void f() {
2   int i=1; j=10;
3   if(MYTHREAD%2){
4     while(i<10){
5       upc_barrier; i++;
6     }
7   }else{
8     while(j<20){
9       upc_barrier; j++;
10    }
11  }
12 }
```

(d) UPC example

```
1  void f() {
2   ...
3   if(x)
4     collective
5   ...
6   if(!x)
7     collective
8
9  }
```

(e) Not verifiable

Fig. 1: Examples of collective issues and a correct program not verifiable by our analysis.

executed by all threads within the same block. Before the first synchronization, the array `tile` depends on thread IDs at line 7. As the array is shared among threads, they all share the same version after the synchronization. The synchronization at line 10 is conditionally executed depending on `tile[0]`. As this value does not depend on thread ID, there is no possible deadlock. Depending on the driver, the third synchronization at line 12 may lead to either a deadlock or an undefined memory configuration. This bug can be difficult to detect for a programmer in a real code as this is a silent synchronization error.

The code Figure 1d is written in Unified Parallel C (UPC), a PGAS language. The predefined variable `MYTHREAD` specifies thread index. In this code, because of the multi-valued expression at line 3, threads with odd ID will call nine barriers (l.5) while the others will call ten barriers (l.9). Although structurally correct, this code leads to a deadlock. Our analysis reports all collectives in a loop as potentially deadlocking. But note that our analysis would return a false positive if the two loops had the same number of iteration.

A static analysis on the previous codes only detects situations and causes of possible deadlocks. If the codes are executed with only one process, no deadlock can happen. Also, our analysis returns a false positive for some correct but structurally incorrect codes like the example Figure 1e. In addition to our analysis, we use PARCOACH instrumentation of programs explained in [5] to handle such situations.

3 PARCOACH Control-flow Analysis

Our analysis first uses PARCOACH to find all conditionals (flow-divergences) that may lead to the execution of different collective sequences.

PARCOACH static analysis relies on the Parallel Program Control Flow Graph (PPCFG) representation of a program [5]. The PPCFG is a program control-flow graph (CFG) where nodes are basic blocks and edges represent the possible flow of control between nodes. To build the PPCFG, PARCOACH reduces each function control-flow graph and replaces each callsite by the callee reduced CFG. Then, with a graph traversal of the PPCFG, PARCOACH computes the possible execution order r of each collective c and the *iterated post-dominance frontier* (PDF^+) for collectives of the same type and order. For a set of collectives $C_{r,c}$, $\text{PDF}^+(C_{r,c})$ corresponds to the control-flow divergences that may result in the execution or non-execution of a collective in $C_{r,c}$. Note that to handle communicators in MPI programs, PARCOACH analyses the program for each communicator separately.

4 Multi-Valued Expression Detection

PARCOACH finds all conditionals potentially leading to different sequences of collectives but reports false positives when conditionals do not depend on a multi-valued expression. This section presents our multi-valued expressions detection.

Enhanced SSA Our analysis is based on the Static Single Assignment (SSA) form of the program. In SSA, variables are defined exactly once. Variables assigned in multiple statements are renamed into new instances, one per statement. This makes explicit def/use chains. When multiple control-flow paths join in the CFG, renamed variables are combined with a ϕ-function into a new variable instance. To capture control-flow dependences we compute an enhanced SSA where ϕ-functions are augmented with their predicates: $\phi(y_1, ..., y_k)$ is transformed into $\phi(y_1, ..., y_k, p_1, ..., p_k)$ with p_i the conditionals responsible for the choice of the y_i. These conditionals are determined by computing the PDF^+ of each argument y_i of the ϕ-function as in [6].

For C-like programs, variables that can be referenced with their address (*address-taken variables*), are only manipulated through pointers with load and store instructions in the SSA form. To compute def/use chains for address-taken variables, we rely on the principles exposed in flow-sensitive pointer analyses

such as [7, 8]: First a points-to analysis is computed to handle potential aliases among arrays and pointers. Then, each load $q = *p$ is annotated with a function $\mu(o)$ for each variable o that may be pointed-to by p to represent a potential use of o at the load. Likewise, each store $*p = q$ is annotated with $o = \chi(o)$ for each variable o that may be pointed-to by p to represent a potential def of o at the store. There is a special case to consider for shared variables. After synchronization (`#pragma omp barrier` in OpenMP, `syncthreads` in CUDA), shared variables have the same value for all threads. To create a new SSA instance that no longer depends on the value preceding the barrier, synchronizations are annotated with $o = \chi()$ for all shared variables o. Then a context-sensitive *Mod-Ref Analysis* is performed to capture inter-procedural uses and def as described in [9]. The purpose of this analysis is to capture the variables referenced and/or modified in functions through pointers. Each callsite cs is annotated with $\mu(o)$ for each variable o indirectly referenced in the called function. Similarly, each callsite is annotated with $o = \chi(o)$ to generate a new instance of o for each variable indirectly modified in the called function. For each address-taken variable referenced or modified in a function, a χ function is inserted at the beginning of the entry node of the CFG and a μ function is inserted at the end of the exit node of the CFG to represent their initial and final values. Finally, all address-taken variables are converted into an SSA form. This results in an augmented SSA with value and control dependences, and additional statements in SSA describing the effects of pointer manipulations. All possible def/use chains are built inside the SSA notation. This simplifies the construction of a dependence graph.

PDCG: Program Data- and Control-flow Dependence Graph A *program data- and control-flow dependence graph* (PDCG) is built from the enhanced SSA by connecting the def of each variable with its uses, following the rules presented in Figure 2. The PDCG captures inter-procedural dependences (represented by edges between variables from different functions) but its construction only requires to analyze each function once. This graph is used to find all variables/expressions that are multi-valued. To that end, we identify the source statements that generate processes identifier and spread the dependencies following the edges of the PDCG. The first four rules (from `OP` to `STORE`) are based on the work in [7] using similar notations. Our differences are highlighted in grey.

`OP` and `PHI` rules correspond to straightforward data- and control-flow dependences. For an operation $\ell : z = x$ op y, the def of x and y at lines ℓ' and ℓ'' are connected to the def of z at line ℓ. For a ϕ statement $\ell : v_3 = \phi(v_1, v_2, ..., p_i, ..)$, the defs of the old SSA instances v_1 and v_2 at ℓ_1 and ℓ_2 are connected to the def of the new SSA instance v_3 at ℓ. For each predicate p_i, the def of p_i at ℓ_3 is connected to the def of v_3 at ℓ to handle the control-flow dependence.

`LOAD` and `STORE` rules take into account alias information for load and store statements. For a load statement $\ell : q = *p$, the def of each object o at ℓ'' pointed to by p is connected to the def of q at ℓ. We also add a link from the def of p at ℓ' to the def of q at ℓ to denote the dependence of q with the array index. Indeed,

this corresponds to the case where $*p$ is in the form of $A[e]$ with e an expression. If e is multi-valued, then q is multi-valued. Similarly, for each store instruction $\ell : *p = q$ annotated with $[o_2 = \chi(o_1)]$, the defs of q and p are connected to o_2. However, we do not connect o_1 to o_2 since we assume that the old value o_1 is overwritten with o_2 (strong update).

The `CALL` rule handles inter-procedural dependences. At each callsite $\ell_{cs} : r = f(..., p, ...)$, the def of the effective parameter p is connected to the formal parameter q in f. Furthermore, the def of the return value x in f is connected to the def of r at ℓ_{cs}. To handle indirect value-flows for address-taken variables, given a callsite annotated with $[\mu(o_1)]\ [o_2 = \chi(o_1)]$, the def of o_1 in the calling function is connected to o_3, the first def of o in f. Similarly, the last def of o in f denoted o_4, is connected to the def of o_2 at ℓ_{cs}.

Rule	Statement (SSA form)	Edges in the PDCG
	Value Flow Dependence	
OP	$\ell : z = x@\ell'$ op $y@\ell''$	$z@\ell \hookleftarrow x@\ell' \quad z@\ell \hookleftarrow y@\ell''$
PHI	$\ell : v_3 = \phi(v_1@\ell_1, v_2@\ell_2, \ ..., \mathbf{p@\ell_3}, ...)$	$v_3@\ell \hookleftarrow v_1@\ell_1 \quad v_3@\ell \hookleftarrow v_2@\ell_2$ $\mathbf{v_3@\ell \hookleftarrow p@\ell_3}$
LOAD	$\ell : q = *p@\ell'\ [\mu(o@\ell'')]$	$q@\ell \hookleftarrow o@\ell'' \quad \mathbf{q@\ell \hookleftarrow p@\ell'}$
STORE	$\ell : *p@\ell_1 = q@\ell_2\ [o_2 = \chi(o_1@\ell_3)]$	$o_2@\ell \hookleftarrow q@\ell_2$ ~~$\mathbf{o_2@\ell \hookleftarrow o1@\ell_3}$~~ $\quad \mathbf{o_2@\ell \hookleftarrow p@\ell_1}$
CALL	$\ell_{cs} : r = f(..., p@\ell_1, ...)\ [\mu(o_1@\ell_2)]$ $[o_2 = \chi(o_1)]$ $f(..., q@\ell_3, ...)\ \{$ $\quad [o_3@\ell_4 = \chi()]\ ...\ [\mu(o_4@\ell_5)]\ return\ x@\ell_6\ \}$	$q@\ell_3 \hookleftarrow p@\ell_1\ \ r@\ell_{cs} \hookleftarrow x@\ell_6$ $o_3@\ell_4 \hookleftarrow o_1@\ell_2$ $o_2@\ell_{cs} \hookleftarrow o_4@\ell_5$
	Optimization	
PHI ELIM	$\ell : *p@\ell_1 = q@\ell_2\ [o_2 = \chi(o_1@\ell_3)]$ $\ell' : *p@\ell_1 = q@\ell_2\ [o_3 = \chi(o_1@\ell_3)]$ $\ell'' : o_4 = \phi(o_2@\ell, o_3@\ell', pred@\ell_4)$	$o_4@\ell'' = \text{fuse}(o_2@\ell, o_3@\ell', o_4@\ell'')$ $\text{remove}(o_4@\ell'' \hookleftarrow pred@\ell_4)$
RESET	$\ell_{cs} : \text{reset}(\text{buf}@\ell_1, ...)\ [\mu(o_1@\ell_2)]$ $[o_2 = \chi(o_1)]$ $\text{reset}(\text{buf}@\ell_3, ...)\ \{$ $\quad [o_3@\ell_4 = \chi()]\ ...\ [\mu(o_4@\ell_5)]\ \}$	$\text{remove}(o_2@\ell_{cs} \hookleftarrow o_4@\ell_5)$
	Collective Checking	
COND	$\ell : collective(...)$ with execution order r For all BB $\in \text{PDF}^+(C_{r,collective@\ell})$ matching: ... br cond$@\ell'$, label1, label2	$collective@\ell \hookleftarrow cond@\ell'$

Fig. 2: Building Rules for the PDCG.

`PHI ELIM` and `RESET` both correspond to edge removal optimizations. After augmenting ϕ-nodes with their predicates, false control dependences can appear if every operand of a ϕ-node denotes the same value. This occurs in particular when considering two identical function calls in two branches of an `if..then..else` construct. Even if these two calls use the same single-valued parameters, the returned value will still depend on the predicate of the conditional (augmented SSA). To tackle this issue, the `PHIELIM` rule fuses such ϕ-nodes with their operands and disconnects the predicates. In distributed memory, after a value-sharing collective such as an all-to-all collective, the communicated buffer has the same value for all processes. This implies that this buffer does not

depend on processes ranks after such collective, whatever its rank-dependence before the collective. To handle this situation, the `RESET` rule disconnects the path from the old SSA instance of the buffer to its new SSA instance after a value-sharing collective ($o_1@\ell_2 \hookrightarrow o_3@\ell_4 \hookrightarrow o_4@\ell_5 \hookrightarrow o_2@\ell_{cs}$). The same rule applies to any value-sharing collective where all processes receive the same result such as `MPI_Allreduce` or `MPI_Broadcast`.

Finally, to detect collectives that may not be executed by all processes/threads, we rely on PARCOACH analysis. Each collective c of execution order r is connected to all conditionals in $\mathrm{PDF}^+(C_{r,c})$ (`COND` rule).

5 Collective Errors Detection

We use the PDCG to track values and nodes that depend on processes identifiers, flooding the graph from seed functions returning IDs or variables allowing tasks to identify themselves: `MPI_Comm_rank` and `MPI_Group_rank` in MPI, `omp_get_thread_num` for OpenMP. In UPC and CUDA, the seed is a variable: `MYTHREAD` and `threadIdx.*`. We use the dependence information from the PDCG to filter out single-valued conditionals from the PDF^+ of potentially unsafe collectives and thus reduce the number of false positives in PARCOACH. The augmented SSA takes into account value and control dependencies and the points-to analysis provides the dependencies through aliases. Note that thanks to the PDCG, our analysis can be path sensitive: An expression may be multi-valued or not, depending on the preceding calling context.

Algorithm 1 describes our whole collective errors detection. Step 1 represents PARCOACH control-flow analysis (see sec. 3) while steps 2 and 3 respectively detect multi-valued expressions and build the PDCG (see sec. 4). Finally, step 4 filters out single-valued conditionals and outputs warnings for potential collective errors.

Example Figures 3a and 3c show the enhanced SSA for the MPI code Figure 1a. The call to `MPI_Comm_rank` is annotated with a χ function to denote the indirect definition of object `o'1` pointed-to by `r`. This generates a new SSA instance denoted `o'2`. Then the object `o'2` pointed-to by `r` is loaded in `reg0`. Depending on whether its value is odd or even, the execution flows either to the basic block labelled `if.then` or the basic block labelled `if.else`. These two control-flow paths join at basic block `if.end` and a ϕ-function is inserted to combine the values of `reg1` and `reg2` into variable `n`. The predicate `cmp1` is added to the ϕ-function to indicate its value depends on `cmp1`.

Figure 3b shows the PDCG corresponding to this example. Rectangle nodes represent collectives. Diamond and circle nodes respectively represent definitions of address-taken and *top-level variables* (variables never referenced by their address). The seed function is `MPI_Comm_rank` line 7 and the first multi-valued object is `o'2`. All library functions have mocked-up CFGs, tagging output val-

// STEP 1. PARCOACH Control-flow Analysis
Input: PPCFG, **Output:** $\bigcup_{r,c} O_{r,c}$ ▷ Create the set $O_{r,c} = PDF^{+}(C_{r,c})$ for each collective name c of execution order r

// STEP 2. Multi-Valued Expression Detection
Input: SSA, **Output:** eSSA ▷ Build an enhanced SSA (eSSA) that captures data- and control-flow dependencies

// STEP 3. PDCG Construction
Input: eSSA, seeds, **Output:** PDCG ▷ Build a PDCG to find all multi-valued expressions and variables from seed functions and variables

// STEP 4. Filter-out single-valued conditionals
Input: $\bigcup_{r,c} O_{r,c}, PDCG$, **Output:** O
for *c in collective names of execution order r* **do**
  for *each node n in* $O_{r,c}$ **do**
    if *n is single-valued in PDCG (there is an edge between c and n in the PDCG)* **then**
      $O_{r,c} \leftarrow O_{r,c} - n$ ▷ remove the node
  end
  $O \leftarrow O \cup (c, \{O_{r,c}\})$
end
Output nodes in O as warnings
return O

Algorithm 1: Collective Error Detection.

ues as multi-valued when necessary. The graph highlights the rank-dependent path from `o'2` to `MPI_Reduce` in `g` passing through the conditional `cmp2` in `f`.

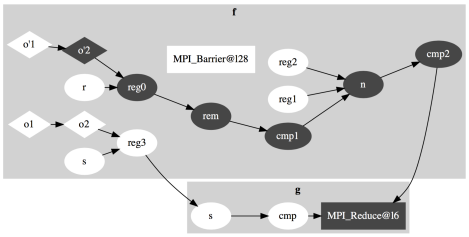
In this example, the execution of `MPI_Reduce` depends on the value of `cmp` in `g` and the call to `g` depends on the value of `cmp2` in `f`. Hence, `MPI_Reduce`@ℓ_6 is connected to `cmp` and `cmp2`. However, there exists no path from `o'2` to `cmp` as `cmp` does not depend on processes ranks. The execution of `MPI_Barrier` is not governed by any conditional. `MPI_Barrier`@ℓ_{28} is then not connected to any node in the graph and it cannot be reached from a seed statement. Since the only collective highlighted in the graph corresponds to `MPI_Reduce` in `g` and only one of the two conditionals governing its execution is highlighted, our new analysis only issues a warning for the multi-valued conditional line 22 in `f` and the call to `MPI_Reduce` in `g`.

6 Related Work

This section summarizes works on dependence analyses and gives an overview of existing tools for collective errors detection in parallel programs.

```
1  define void f() {
2  entry:
3    s = alloca_o; // object o1
4    r = alloca_o';// object o'1
5    MPI_Comm_size(com, s);
           [μ(o1)]
6    [o2 = χ(o1)]
7    MPI_Comm_rank(com, r);
           [μ(o'1)]
8    [o'2 = χ(o'1)]
9    reg0 = load r [μ(o'2)]
10   rem = reg0 % 2;
11   cmp1 = rem != 0;
12   br cmp1, if.then, if.else
13 if.then:
14   reg1 = 1;
15   br label if.end
16 if.else:
17   reg2 = 2;
18   br label if.end
19 if.end:
20   n = φ(reg1, reg2, cmp1)
21   cmp2 = n == 1;
22   br cmp2, if.then2, if.end2
23 if.then2:
24   reg3 = load s; [μ(o2)]
25   g(reg3);
26   br label if.end2
27 if.end2:
28   MPI_Barrier(com);
29   ret void;
30 }
```

(a) Function `f` enhanced SSA.



(b) PDCG.

```
1  define void g(i32 s) {
2  entry:
3    cmp = s > 256
4    br cmp, if.then, if.end
5  if.then:
6    MPI_Reduce(com, ...);
7    br if.end
8  if.end:
9    ret void
10 }
```

(c) Function `g` enhanced SSA.

Fig. 3: Enhanced SSA form of the MPI code Figure 1a and its corresponding PDCG.

6.1 Dependence Analyses Techniques

Dependence analyses are the cornerstones of many optimizations/analyses in compilers. For instance, dependences are used for *Taint Analysis* [10, 11, 12] to determine how program inputs may affect the program execution and exploit security vulnerabilities, *Information Flow Tracking* [13, 14, 15, 16] to prevent confidential information from leaking, static *Bug Detection* [17, 18] or code optimization and parallelization (e.g. the polyhedral model [19]). One of the difficult issues when computing data dependences is to deal with pointers/memory aliases and non scalar variables (e.g. arrays, structures). In SVF [7] the authors annotate load and store instructions with μ and χ functions to transform address-taken variables into an SSA form. However, they do not take into account the possible dependence of the pointer itself (through an array index for instance) when they build the data dependence graph.

Many of the aforementioned analyses only consider data dependences whereas Slowinska *et al.* [20] showed that omitting control dependences can be a huge source of false negative results. In [21], the authors introduced the concept of *Strict Control Dependences* to reduce the number of false positives in *Taint Analyses* and *Lineage Tracing*. In Parfait [22] the authors propose to extend

ϕ-functions with predicates in order to handle control dependences. However, address-taken variables are not transformed into an SSA form.

6.2 Collective Error Detection Techniques

Static analyses operate on the source code of an application and have the advantage of not requiring execution. They are usually based on model checking and symbolic program execution, limiting their applicability to small and moderate sized applications (the number of reachable states to consider is combinatorial). TASS [23] and CIVL [24] use this approach. They rely on symbolic execution and require source code annotations to detect collective errors in MPI programs. The OpenMP Analysis Toolkit (OAT) [25] uses the same method for OpenMP programs by exploring all program paths under symbolic values. SimGridMC [26]is a model checker for MPI applications. It uses Dynamic Partial Order Reduction and State Equality techniques to face the state space explosion problem. UPC-SPIN [27] generates finite models of UPC programs in the modeling language of the SPIN model checker. For CUDA programs, we can mention GPUVerify [28] that statically checks that all threads execute the same barriers syntactically. Unlike our analysis, the method does not give a precise feedback in case of a potential error. PARCOACH combines an inter-procedural control-flow analysis with a selective instrumentation to find MPI and OpenMP collective errors. The method is limited to control-flow information and returns many false positives. Our new analysis overcomes this limitation and extends the collective verification to other parallel programming models. The method presented by Zhang and Duesterwald in [1] is the closest to our work. It detects synchronization errors with an inter-procedural barrier matching technique for SPMD programs with textually unaligned barriers. Compared to our analysis, this method is only focused on MPI and OpenMP synchronizations and has no pointer analysis.

Unlike static tools, dynamic tools do not report false positives. However, they are dependent on the input data set and may miss errors (false negatives). PARCOACH instruments non verifiable programs to verify if processes/threads are going to call the same collective at a particular step of execution, preventing deadlocks from happening. This instrumentation is similar to what dynamic tools like MUST [29] or UPC-CHECK [30] do. However, the instrumentation starts with the first collectives that may deadlock, avoiding a full instrumentation of programs.

7 Experimental Results

Our analysis is implemented as a pass in the LLVM framework 3.9 integrated into the open source software PARCOACH[5].

Figures 4 and 5 show the impact of our multi-valued expression analysis on PARCOACH. Figure 4 displays the percentage of warnings and conditionals filtered out with our multi-valued analysis compared to the initial PARCOACH

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

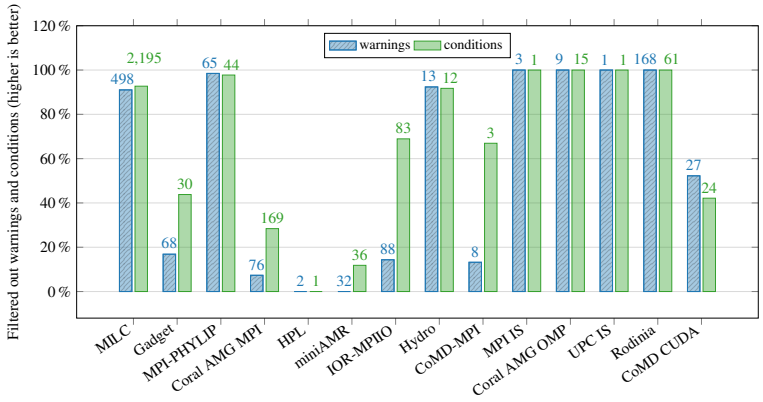


Fig. 4: Percentage of warnings and conditionals filtered by our multi-valued analysis. 100% means that the analysis proves the program is collective error free.

analysis on 3 HPC applications (MILC[6], Gadget[7] and MPI-PHYLIP [31]), 3 mini HPC applications (CoMD and miniAMR from the Mantevo project [32] and Hydro[8]) and 5 widely used benchmarks (HPL[9], IOR[10], AMG[11], NAS IS[12], and the CUDA benchmarks from Rodinia[13]). In the figure, *warnings* are collectives that may lead to deadlocks, and *conditions* correspond to conditionals governing the execution of unsafe collectives. The initial number of warnings and conditionals found by PARCOACH is given at the top of each bar. 100% for a warning bar means that the application is collective error-free (all warnings are removed by our analysis, the code is proved safe), 0% means that our analysis has no impact. For MILC, 91% of the 498 warnings have been removed. PARCOACH now reports 45 warnings. As shown in the figure, about half conditionals are filtered out by our analysis for most applications and all warnings are removed for Coral AMG OMP, MPI IS, UPC IS and Rodinia. Figure 5 gives the percentage of collectives tagged as potentially deadlocking by our analysis. The total number of collectives is given at the top of each bar. In the figure, seven applications have less than 20% of collectives potentially deadlocking.

To highlight the functionality of our analysis, we created a microbenchmark suite containing programs from multiple sources with correct and incorrect use of MPI collectives[14]. We compare the performance of the method presented in

[6] `http://www.physics.utah.edu/~detar/milc/`
[7] `https://wwwmpa.mpa-garching.mpg.de/gadget/`
[8] `https://github.com/HydroBench/Hydro`
[9] `http://www.netlib.org/benchmark/hpl`
[10] `http://www.nersc.gov/research-and-development/apex/apex-benchmarks/ior`
[11] `https://asc.llnl.gov/CORAL-benchmarks/`
[12] `http://www.nas.nasa.gov/software/NPB`
[13] `https://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php`
[14] The microbenchmark suite is available at `https://gitlab.inria.fr/parcoach/microbenchmarks`

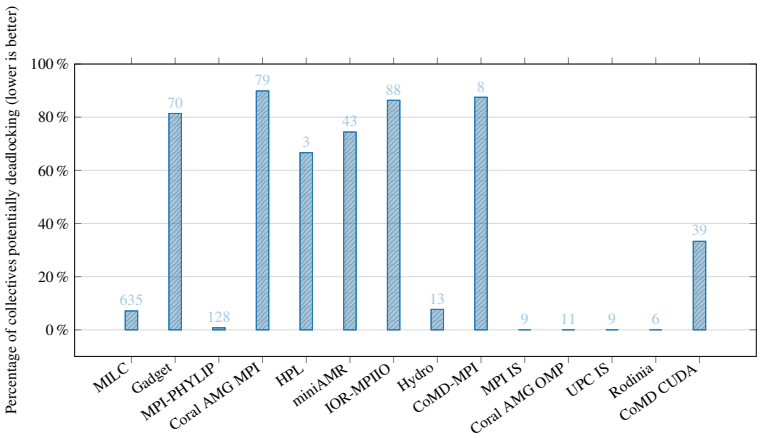


Fig. 5: Percentage of collectives potentially deadlocking.

Table 1: Multi-valued detection comparison between the work in [1] and PARCOACH (PAR.) using our PDCG, SVF and Parfait. FP = false positives, FN = false negative.

Program Name	**Origin**	**Description**	**Deadlock**	**Zhang**	**PARCOACH**		
				***et. al* [1]**	**PDCG**	**SVF**	**Parfait**
field-sensitive	Hydro	Structure with a multi-valued field	no	FP	FP	FP	FP
index-dep	PAR.	Use of an array	yes	✓	✓	FN	✓
phi-cond	PAR.	Control-flow dependence	yes	✓	✓	FN	✓
pointer-instance	PAR.	Figure 1b	yes	✓	✓	✓	FP
pointer-alias	PAR.	Use of aliases	yes	FN	✓	✓	FP
barrierReduce	CIVL	Collective mismatch	yes	FN	✓	✓	✓
barrierScatter	CIVL	Collective mismatch	yes	FN	✓	✓	✓
BcastReduce_bad	CIVL	Collective mismatch	yes	FN	✓	✓	✓
mismatch-barrier	PAR.	Collective mismatch	yes	✓	✓	✓	✓
mismatch_barrier_com	PAR.	Collective mismatch	yes	✓	✓	✓	✓
mismatch_barrier_nb	PAR.	Collective mismatch	yes	✓	✓	✓	✓
MPIexample	PAR.	Figure 1a	yes	FN	✓	FN	FN
noerror_barrier	PAR.	Correct usage of barrier	no	✓	✓	✓	✓
not_verifiable	PAR.	Figure 1e	no	FP	FP	FP	FP
loop_barrier	PAR.	Figure 1d	yes	✓	✓	✓	✓

[1] and PARCOACH using our multi-valued analysis (PDCG), SVF and Parfait. Table 1 shows the results. Our analysis always detect collective errors compared to the others. For the remaining false-positive results, a more precise dependence analysis is required. This is left for future work.

8 Conclusion

This article presents a new static/dynamic method to verify that a parallel program has structurally correct collectives. The analysis resorts to an inter-procedural static analysis that can prove in some cases that a program is free of collective error. The method has been applied successfully on different languages and is implemented in PARCOACH. Experiments show that our analysis leads to significant improvement over existing PARCOACH. Furthermore, through a

more precise use of alias and control dependences, our static analysis outperforms existing data-flow analyses bringing additional preciseness (removing spurious dependencies) and correctness (adding missing dependencies).

References

[1] Zhang, Y., Duesterwald, E.: Barrier matching for programs with textually unaligned barriers. PPoPP, ACM (2007) 194–204

[2] Jakobsson, A., Dabrowski, F., Bousdira, W., Loulergue, F., Hains, G.: Replicated synchronization for imperative bsp programs. Procedia Computer Science **108** (2017) 535 – 544 International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland.

[3] Aiken, A., Gay, D.: Barrier inference. In: Proc. ACM SIGPLAN-SIGACT Symp. on Principles of Prog. Lang. POPL, ACM (1998) 342–354

[4] Saillard, E., Carribault, P., Barthou, D.: PARCOACH: combining static and dynamic validation of MPI collective communications. IJHPCA **28**(4) (2014) 425–434

[5] Huchant, P., Saillard, E., Barthou, D., Brunie, H., Carribault, P.: PARCOACH Extension for a Full-Interprocedural Collectives Verification. In: Second International Workshop on Software Correctness for HPC Applications. (2018)

[6] Scholz, B., Zhang, C., Cifuentes, C.: User-input dependence analysis via graph reachability. (2008)

[7] Sui, Y., Xue, J.: SVF: Interprocedural Static Value-flow Analysis in LLVM. CC (2016) 265–266

[8] Hardekopf, B., Lin, C.: Flow-sensitive Pointer Analysis for Millions of Lines of Code. CGO (2011) 289–298

[9] Sui, Y., Ye, D., Xue, J.: Detecting Memory Leaks Statically with Full-Sparse Value-Flow Analysis. IEEE Trans. Softw. Eng. **40**(2) (2014) 107–122

[10] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. SIGPLAN Not. **49**(6) (2014) 259–269

[11] Tripp, O., Pistoia, M., Fink, S.J., Sridharan, M., Weisman, O.: TAJ: Effective Taint Analysis of Web Applications. PLDI (2009) 87–97

[12] Shankar, U., Talwar, K., Foster, J.S., Wagner, D.: Detecting Format String Vulnerabilities with Type Qualifiers. SSYM (2001)

[13] Denning, D.E., Denning, P.J.: Certification of Programs for Secure Information Flow. Commun. ACM **20**(7) (1977) 504–513

[14] Heintze, N., Riecke, J.G.: The SLam Calculus: Programming with Secrecy and Integrity. POPL (1998) 365–377

[15] Yin, H., Song, D., Egele, M., Kruegel, C., Kirda, E.: Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. CCS (2007) 116–127

[16] Sabelfeld, A., Myers, A.C.: Language-based Information-flow Security. IEEE J.Sel. A. Commun. **21**(1) (2006) 5–19

[17] Laguna, I., Schulz, M.: Pinpointing Scale-dependent Integer Overflow Bugs in Large-scale Parallel Applications. SC (2016) 19:1–19:12

[18] Ye, D., Sui, Y., Xue, J.: Accelerating Dynamic Detection of Uses of Undefined Values with Static Value-Flow Analysis. CGO (2014) 154:154–154:164

[19] Feautrier, P.: Dataflow Analysis of Array and Scalar References. International Journal of Parallel Programming **20**(1) (1991) 23–53
[20] Slowinska, A., Bos, H.: Pointless Tainting?: Evaluating the Practicality of Pointer Tainting. EuroSys (2009) 61–74
[21] Bao, T., Zheng, Y., Lin, Z., Zhang, X., Xu, D.: Strict Control Dependence and Its Effect on Dynamic Information Flow Analyses. ISSTA (2010) 13–24
[22] Cifuentes, C., Scholz, B.: Parfait: Designing a Scalable Bug Checker. In: Proceedings of the 2008 Workshop on Static Analysis. SAW (2008) 4–11
[23] Siegel, S.F., Zirkel, T.K.: Automatic formal verification of mpi-based parallel programs. SIGPLAN Not. **46**(8) (February 2011) 309–310
[24] Siegel, S.F., Zheng, M., Luo, Z., Zirkel, T.K., Marianiello, A.V., Edenhofner, J.G., Dwyer, M.B., Rogers, M.S.: Civl: the concurrency intermediate verification language. In: SC. (Nov 2015) 1–12
[25] Ma, H., Diersen, S.R., Wang, L., Liao, C., Quinlan, D., Yang, Z.: Symbolic Analysis of Concurrency Errors in OpenMP Programs. In: PARCO. Volume 00 of ICPP. (2013) 510–516
[26] Pham, T.A., Jeron, T., Quinson, M.: Verifying MPI Applications with SimGridMC. In: Correctness. (2017)
[27] Ali, E.: UPC-SPIN: a framework for the model checking of UPC programs. (2011)
[28] Betts, A., Chong, N., Donaldson, A., Qadeer, S., Thomson, P.: GPUVerify: A Verifier for GPU Kernels. OOPSLA, ACM (2012) 113–132
[29] Hilbrich, T., de Supinski, B.R., Hänsel, F., Müller, M.S., Schulz, M., Nagel, W.E.: Runtime MPI Collective Checking with Tree-based Overlay Networks. EuroMPI (2013) 129–134
[30] Coyle, J., Roy, I., Kraeva, M., Luecke, G.R.: UPC-CHECK: a scalable tool for detecting run-time errors in Unified Parallel C. Computer Science - Research and Development **28**(2) (2013) 203–209
[31] Ropelewski, A.J., Nicholas, Jr, H.B., Gonzalez Mendez, R.R.: MPI-PHYLIP: Parallelizing Computationally Intensive Phylogenetic Analysis Routines for the Analysis of Large Protein Families. PLOS ONE **5**(11) (2010) 1–8
[32] Heroux, M.A., Doerfler, D.W., Crozier, P.S., Willenbring, J.M., Edwards, H.C., Williams, A., Rajan, M., Keiter, E.R., Thornquist, H.K., Numrich, R.W.: Improving Performance via Mini-applications. (2009)