Static Validation of Barriers and Worksharing Constructs in OpenMP Applications

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

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

Abstract. The OpenMP specification requires that all threads in a team execute the same sequence of worksharing and `barrier` regions. An improper use of such directive may lead to deadlocks. In this paper we propose a static analysis to ensure this property is verified. The well-defined semantic of OpenMP programs makes compiler analysis more effective. We propose a new compile-time method to identify in OpenMP codes the potential improper uses of barriers and worksharing constructs, and the execution paths that are responsible for these issues. We implemented our method in a GCC compiler plugin and show the small impact of our analysis on performance for NAS-OMP benchmarks and a test case for a production industrial code.

1 Introduction

OpenMP is a popular parallel programming model for shared memory machines. While OpenMP aims at making parallel programming easier, there are a number of improper uses of worksharing constructs and barriers that are not statically detected by compilers and may lead to deadlock or unspecified behavior. Indeed, the OpenMP specification requires that all threads of a team must execute the same sequence of worksharing constructs and barriers [16]. However in practice no error occurs when all threads of a team do not execute exactly the same barrier. That is why we authorize threads synchronizations with different barriers and defined two verbosity levels (0 and 1) defining soft and hard barriers verifications. Throughout the rest of the paper, examples are presented with verbosity level 0.

To show the difficulty to enforce this constraint in OpenMP codes, consider the motivating examples in Figure 1. In function `f` of Listing 1.1, each thread may or may not encounter the `single` construct line 9, depending on the control flow (line 6). According to the OpenMP specification, all threads in a team should encounter the same `single`, or none of them. However, compiling this code and executing it does not lead to a syntactic error but to a deadlock. Indeed, if the result of the conditional is not the same among all threads, the first barrier executed will be for some threads the implicit barrier line 12 (end of `single`) while for others, it will be the explicit barrier line 14. Then the first group of threads will stop at the explicit barrier line 14 while the second group will stop at the barrier related to the end of the `parallel` region. Finally, the first set of threads will be released and eventually deadlock at this last barrier. Note that if

Listing 1.1.

```
1  void f( ){
2    if(...)
3    {
4      #pragma omp parallel
5      {
6        if(...)
7        {
8          /*...*/
9          #pragma omp single
10         {
11           /*...*/
12         }
13       }
14       #pragma omp barrier
15       /*...*/
16     }
17   }
18 }
```

Listing 1.2.

```
1  void f() {
2    /*...*/
3    #pragma omp barrier
4    return;
5  }
6
7  int main(  ) {
8    int r;
9    #pragma omp parallel private(r)
10   {
11     r=...;
12     if( r == 0 )
13       f();
14   }
15   exit(0);
16 }
```

Fig. 1. Examples with Deadlock Situations

we modify this example by adding an `else` statement with another `single`, the code is still potentially erroneous since all threads should encounter the same `single`. A more complex case appears Listing 1.2. A deadlock can occur at the end of the parallel region of function `main` because of the conditional line 12. Depending on the control flow the barrier in `f` may be not encountered by all threads. The error is more difficult to detect and an interprocedural analysis is required. This illustrates the fact that the machine state does not help to identify the cause of deadlocks (in these two examples, conditionals).

This paper proposes a new compile-time technique to detect potential improper uses of worksharing constructs and barriers in applications parallelized with OpenMP. The main advantage of our method is to highlight the statements responsible for the execution path potentially leading to future deadlocks or unspecified behaviors. This contribution is an adaptation and transposition of the work presented in [12] for checking MPI applications with respect to barriers. The OpenMP application is checked function per function, using intra-procedural analysis. Each function of a program is said to be correct if all threads of the same team (entering the function or created in the function) have the same sequence of worksharing regions and depending of the verbosity level, the same sequence of barriers (verbosity 1) or the same number of barriers (verbosity 0). An inter-procedural analysis complements the analysis for checking the whole application. This paper makes the following contributions:

– Analysis of barriers and worksharing constructs that may lead to deadlocks, identification of the control-flow that may be responsible for these situations;
– Consideration of the OpenMP specification and the practice through two verbosity levels;
– Full implementation inside a production compiler; experimental results on different benchmarks and applications.

The outline of the paper is the following. Section 2 provides a summary of existing debugging tools for OpenMP programs. Section 3 defines the problem statement, describes the program representation we use and presents our compile-time analysis. Section 4 details experimental results before concluding in Section 5.

2 Related Work

OpenMP applications are prone to concurrency errors such as data races and deadlocks. Debugging tools generally check the correctness of OpenMP programs either at compile-time or during execution of a program, both methods having advantages and inconveniences. This section summarizes some existing tools to detect data races and deadlocks in OpenMP applications.

The well-defined semantics of OpenMP makes static analyses common to check the correctness of OpenMP applications. Several static approaches exist: First we can mention the OpenMP Analysis Toolkit [9] (OAT) that uses symbolic analysis to detect concurrency errors. It relies on the ROSE compiler infrastructure to encode every parallel region into Satisfiability Modulo Theories (SMT) formulae. Those formulae are then solved with a SMT-solver like Yices [17]. OAT terminates its analysis by instrumenting the source code with fault injection technique to confirm the reported errors. OmpVerify [1] is a static tool integrated in Eclipse IDE using the polyhedral model to detect data races in OpenMP parallel loops. This tool is restricted to program fragments called Affine Control Loops but it has the advantage of reporting accurate errors to the user. Lin [8] describes a concurrency analysis technique to detect whether two statements will not be executed concurrently by different threads in a team. The method is an intraprocedural analysis based on phase partioning using an OpenMP Control Flow Graph (OMPCFG) that models the transfer of control flow in an OpenMP program. Similarly, Zhang *et al.* [18] use a concurrency analysis to detect unaligned barriers in OpenMP C programs. This inter-procedural method consists in four phases: A CFG construction to model the various OpenMP constructs, a barrier matching to find threads barriers that synchronize together, a program division into phases (sequence of basic blocks separated by barriers) and an aggregation of phases with matching barriers. Any two basic blocks from the same aggregated phase are said to be concurrent. Although quite close to our analysis this work differs from us in several points. Unlike Zhang *et al.*, our analysis is language independent and verifies woksharing-construct placements in a program. To detect possible deadlocks we use the graph representation defined in [8]. Potential errors are automatically returned to the user with the line of the erroneous conditionals by a simple analysis of the OMPCFG. Thus the user knows exactly what can cause a deadlock and correct it. For the verification of the whole program Zhang *et al.* export a barrier tree when we only need an integer defining the minimal number of possible barriers encountered in a function. Then a simple callgraph traversal points out the possible sources of deadlocks in the whole program. However both methods could complement each other. Detection can also be done by compilers like GCC when lowering the OpenMP constructs to GOMP function calls [14]. Indeed, GCC issues a warning for wrong nested parallelism, typically a barrier in a single region. Like all static tools, our method has the advantage of not requiring execution of the program but can produce false positives.

Among dynamic tools we can mention the Adaptative Dynamic Analysis Tool [6] (ADAT) and RaceStand [5, 10] for focused data races detection and Intel Thread Checker [11, 4] and Sun Thread Analyzer [13] for both data races and deadlocks detection. ADAT is a data races detection tool using classification and adaptation mechanisms. The tool creates a pseudo-instrumented source code and an Engine Code Property Selector (ECPS) table and then transforms the pseudo-instrumented source code into an executable by using the ECPS table information. With a C compiler supporting OpenMP, the instrumented source code is compiled and executed to detect data races. RaceStand by GNU utilizes an on-the-fly dynamic monitoring approach to detect data races and has recently improved its check with a dynamic binary instrumentation technique based on Pin software framework. This tool detects the existence of races and locates races between two accesses not causally preceded by other accesses also involved in races (first races) for each shared variable in a program. Intel Thread Checker and Sun Thread Analyzer both require an application instrumentation and trace references to memory and synchronization operations during the application execution. Sun Thread Analyzer necessicates program recompilation with the Sun compilers. To find data races the program must be executed with two or more threads. Unlike Sun Thread Analyzer, Intel Thread Checker does not depend on the number of threads used. It dynamically detects data races using a projection technology which exploits relaxed OpenMP programs. More precisely, the projection technology checks the data dependency of accesses to shared variables using sequentially traced information. But Intel Thread Checker does not consider OpenMP programs specifications and can therefore report false positives. Li *et al.* present in [7] an online-offline model to test the correctness of every OpenMP parallel region. The online correctness testing model is used to find parallel regions with incorrect execution results (not corresponding to serial execution results), identify all places that caused errors (directives used improperly or located wrongly) and correct them. Then the offline correctness testing model tests the correctness of regions with corrected directives. Compared to dynamic tools that detect a deadlock when it occurs, our static analysis prevents programs from deadlocking (the program is stopped whenever a deadlock situation is detected). Moreover our method is not limited to the input dataset of a run. Indeed, even if dynamic tools return no false positive, they can miss errors as they are correlated to one execution of a program.

We proposed in [12] a combining method to detect misuse of MPI collective operations in MPI programs. MPI processes must have the same sequence of collective operations otherwise a deadlock may occur. Restriction on MPI collective operations is the same as restrictions on barriers and worksharing regions in OpenMP programs. Thus we adapted this work to detect potential deadlocks in OpenMP programs. Potential deadlocks due to wrong synchronizations as well as worksharing regions are automatically detected in each function of a program and then errors considering the whole program are reported by an inter-procedural analysis. To our knowledge our analysis is the first intra- and inter-procedural analysis that verifies that all OpenMP tasks encounter the same worksharing regions.

3 Checking OpenMP Directives and Control Flow

In OpenMP programs, the threads of a team can synchronize through the `#pragma omp barrier` directive or at an implicit barrier at the end of worksharing regions (unless a `nowait` clause is specified). Worksharing constructs distribute the execution of the associated region among the threads of a team [16]. Worksharing constructs are loop, sections, single and workshare constructs. The OpenMP specification gives some restrictions to barriers and worksharing constructs. Indeed, each barrier/worksharing region must be encountered by all threads in a team or by none at all, unless cancellation has been requested for the innermost enclosing parallel region ([16] Sections 2.7, p.53 and 2.12.3, p.124). However, due to the control flow inside an OpenMP program, the threads may execute different execution paths with different numbers of barriers and worksharing regions. Such behavior can lead to a deadlock or unspecified behaviors.

The principle of the static analysis we propose is the following. For each function of the code, we check that for all threads entering the function and for all teams created within it, the same number of barriers are executed, whatever the execution path taken by the threads. If the number of barriers may depend on the control flow, the control structures responsible for this are shown with a warning. This is a conservative approach, since we do not check that the conditional of an `if` statement for instance is dependent on the ID of the threads. Moreover, we check that worksharing constructs may not be conditionally executed, potentially leading to unspecified behaviors. This intra-procedural analysis on barriers and worksharing constructs is complemented by a simple inter-procedural analysis: User-defined functions are subsumed by the number of worksharing constructs and barriers executed by the entering threads. This captures all potential improper uses of barriers and worksharing constructs.

The program to analyze is represented using the OMPCFG intermediate representation, briefly described in the following section. Then the intra- and inter-analyses are presented.

3.1 Intermediate Representation: OMPCFG

The *control-flow graph* (CFG) is an intermediate representation of code, used by almost all compilers. The CFG is a directed graph where nodes are basic blocks (straight sequence of code) and edges are potential flow of control between nodes. Lin [8] extended the notion of CFG to a representation for parallel OpenMP programs, called OMPCFG. Each node of the OMPCFG represents a basic block (basic nodes) or an individual block containing an OpenMP directive (directive nodes). In the OMPCFG, implicit barriers are made explicit and each combined parallel worksharing construct is separated into a `nowait` worksharing construct nested in a parallel region. Moreover the OMPCFG has a single *Entry* and single *Exit* nodes. New edges are inserted between basic nodes and directive nodes according to OpenMP semantics. As a result, the `master` directive is represented as a conditional. Table 1 lists the OpenMP directives and their corresponding directive node in the OMPCFG. Note that Lin also adds edges from the end construct directive to the begin construct directive nodes denoted as construct edges. These edges are not considered here as they do not reflect any control flow.

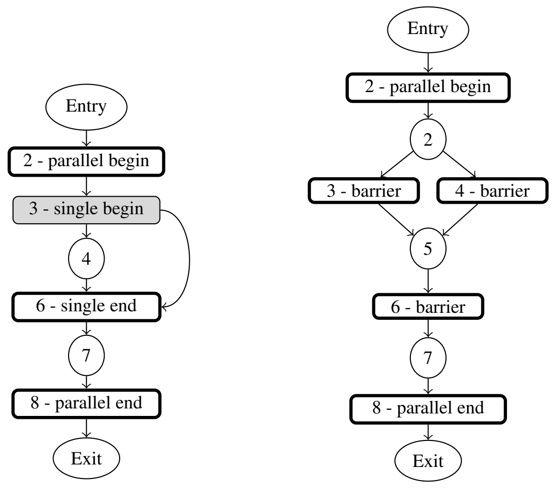
Table 1. Directive nodes in the OMPCFG

Directive name	Control flow	Worksharing construct
parallel, critical, atomic, section, barrier, ordered, task, taskwait, taskyield	linear	
master	if/else	
for, single	if/else	$*$
sections, workshare	switch/case	$*$

Figure 2 shows examples of OMPCFG. All directive nodes containing a `barrier` are represented as thick nodes and all directive nodes containing a worksharing construct are colored in gray. Directive nodes containing a `parallel` construct are considered as barriers but are not considered in our Algorithms. Out of clarity, implicit barriers at the end of worksharing and parallel regions are not designated by barriers but by *`region-name`* `end`.

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

(a) Source code



(b) OMPCFG (c) OMPCFG with barriers

Fig. 2. Example of a simple code (a) with its corresponding OMPCFG (b) and an OMPCFG containing barriers (c)

This representation is the base of our compiler analysis. GCC uses a graph representation similar to the OMPCFG from version 4.2.

3.2 Intra-Procedural Analysis

This section details the static verification of barriers and worksharing constructs for each function of a program. We define two levels of verbosity for barriers verification:

level 0 that returns warnings only if there may be an execution error and level 1 that returns warnings in strict accordance with the specification.

For the verbosity level 0, we identify barrier statements that synchronize together. To that purpose, we introduce a number, the *sequential order*, counting the number of barriers traversed before reaching a barrier. This number is assigned to each node in the OMPCFG. Two nodes with different sequential order are sequentially ordered thanks to barriers. This number is 0 for nodes before the first barrier (including the node with the first barrier), 1 for nodes reached after one barrier and so on. When multiple paths exist, nodes can have multiple numbers, at most the number of barriers in the function. Loop backedges are removed to have a finite numbering. A function is not correct if there are nodes with multiple orders. These nodes correspond to possible control-flow divergence leading to deadlocks. In Zhang *et al.* [18], this notion of sequential order corresponds to phases, computed through an inter-procedural liveness analysis and a barrier aggregation step. While both methods can be used for our goal, our approach is simpler, more adapted to the verification of barriers. The computation of the execution order uses an algorithm adapted from the algorithm 1 in our previous work [12] on MPI verification. This algorithm detects possible control-flow divergence leading to a deadlock in a MPI barrier. MPI barriers are numbered by so-called *execution ranks* (similar to sequential order here). MPI barriers of same execution rank r are put into a set $C_{r,c}$ as matching MPI barriers. c is used to differenciate MPI collective operations names. In our case, only barriers are considered so the c is useless and only C_r sets are created. Algorithm 1 is an adaptation of this method for OpenMP barriers, from line 6 to line 12. Barriers with multiple sequential orders are put in the set C_r with r corresponding to their maximal sequential order. For example the OMPCFG Figure 2(c) contains three explicit barriers nodes 3, 4 and 6 and one implicit barrier node 8. The sequential order for nodes 3 and 4 is 0, for node 6 , 1 and for node 8, 2. The algorithm computes $C_0 = \{3,4\}$, $C_1 = \{6\}$ and $C_2 = \{8\}$.

For the verbosity level 1, we verify each barrier is encountered by all threads of a team. This is described from line 14 to line 16 in Algorithm 1.

The algorithm takes the OMPCFG of the current function and the verbosity level as input parameters and outputs a message error for conditional nodes that may lead to a deadlock in a barrier (set S). The core of the algorithm is based on the post-dominance frontier [2], used in a previous paper in the context of MPI collectives verification [12]: The postdominance frontier of a node u of the OMPCFG (denoted as $PDF(u)$) 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, $PDF(u) = \{v \mid \exists\, w \in SUCC(v), u \underline{\gg} w \text{ and } u \not\gg v\}$. 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 the exit node that does not go through u. This concept is extended to a set of nodes N: $PDF(N) = \bigcup_{u \in N} PDF(u)$ and to the notion of iterated postdominance frontier PDF^+ defined as the transitive closure of PDF, when considered as a relation [2]. If barriers with the same sequential order n have a non-empty PDF^+ set, then some threads may not perform the n^{th} synchronization. Due to the representation of all worksharing constructs (as if/else or switch), barriers inside these worksharing constructs are detected as incorrect.

Algorithm 1 OpenMP Intra-procedural Control-flow Analysis

```
 1: function FUNCTION_VERIFICATION($f, \upsilon$)              ▷ $f$: a function of the application
 2:                                                             ▷ $\upsilon$: level of verbosity
 3:     Compute $G = (V, E)$ the OMPCFG of $f$
 4:     $S \leftarrow \emptyset$, $S' \leftarrow \emptyset$     ▷ Output sets: Conditional nodes
 5:     if $\upsilon = 0$ then                                  ▷ level 0 of verbosity
 6:         Remove loop backedges in $G$ and Compute sequential order of all nodes
 7:         for $n = 0..\max($ sequential order $(G))$ do
 8:             for barriers of sequential order $n$ do
 9:                 $C_n \leftarrow \{u \in V | u$ of order $n\}$
10:                 $S \leftarrow S \cup PDF^+(C_n)$
11:             end for
12:         end for
13:     else                                                    ▷ level 1 of verbosity
14:         for $u \in V$ s.t. $u$ contains an explicit barrier do
15:             $S \leftarrow S \cup PDF^+(u)$
16:         end for
17:     end if
18:
19:     for $u \in V$ s.t. $u$ contains a worksharing construct do
20:         $S' \leftarrow S' \cup PDF^+(u)$
21:     end for
22:     Output nodes in $S'$ and $S$ as warnings
23: end function
```

The lines 19 to 21 of the algorithm detect if worksharing constructs may not be executed by all threads of a team. For each node u containing a worksharing construct, we compute the iterated postdominance frontier of u. If the $PDF^+(u)$ is not empty then some threads may execute the construct while others may avoid it. The set of nodes detected are put in the set S' for warnings.

Lemma 1. *Algorithm 1 is correct if it detects all deadlock situations due to* `barrier` *and worksharing regions.*

Proof. The levels of verbosity enable a strict verification of barriers in compliance with the specification. In that purpose Algorithm 1 detects if all threads of a team have strictly the same sequence of barriers. A soft verification is also possible. The algorithm then verifies all threads of a team encounter the same number of barriers. The proof has been done in [12]. Then Algorithm 1 computes the set S' of control-flow nodes that have execution paths with different number or type of worksharing constructs from the node to the *Exit* node. We prove that nodes in S' correspond exactly to the nodes that lead to a deadlock.

As an example, the first OMPCFG Figure 3 contains one explicit barrier (node 8), two implicit barriers (nodes 7 and 10) and one worksharing construct: `single` (node 5). Algorithm 1 computes sequential orders. Node 7 is of sequential order 0, node 8 is of sequential orders 0 and 1 and finally node 10 is of sequential orders 1 and 2. Thus

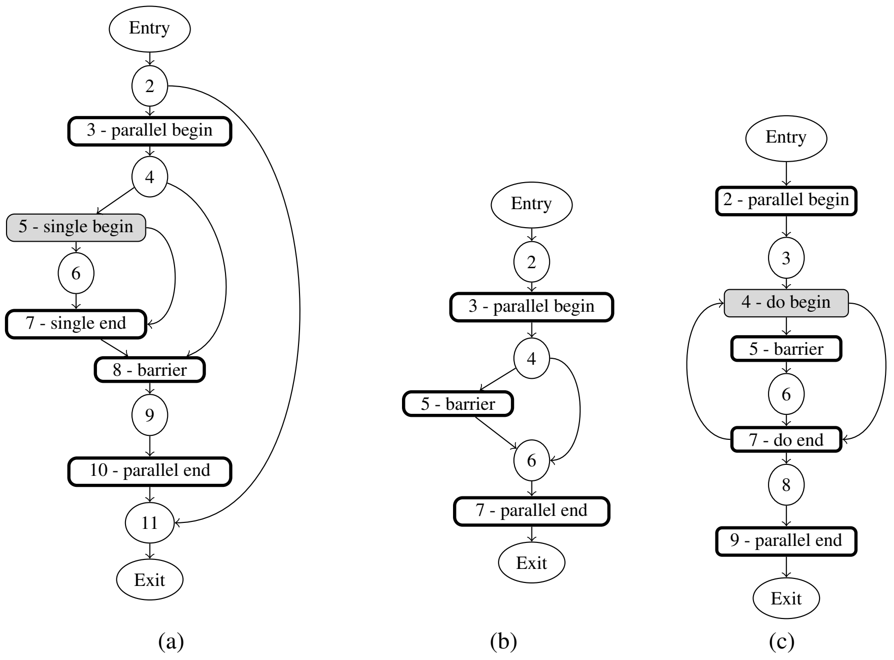


Fig. 3. Functions `f` OMPCFG of Listing 1.1 ((a)) and `main` OMPCFG of Listing 1.2 after function `f` replacement ((b), see Algorithm 2) and an example of an OMPCFG with a loop ((c))

we have $C_0 = \{7\}$, $C_1 = \{8\}$ (node 8 is in C_1 as it has multiple sequential orders) and $C_2 = \{10\}$. $PDF^+(C_1) = \emptyset$ and $PDF^+(C_2) = \emptyset$ but $PDF^+(C_0) = \{4\}$. Node 4 is the only node in the iterated postdominance frontier of node 7 as the conditional node 2 is outside the parallel region. Then the conditional node 4 is returned as the possible cause of a deadlock in a `barrier`. For node 5, $PDF^+(5) = 4$. To sum up for Listing 1.1, a warning is issued for the conditional located in node 4 as potentially leading to different barriers and worksharing constructs sequence among threads. The OMPCFG Figure3(b) contains one explicit barrier node 5 and one implicit barrier node 7. The algorithm computes $C_0 = \{5\}$, $C_1 = \{7\}$ and $PDF^+(C_0) = \{4\}$. Last, the OMPCFG Figure 3(c) contains one worksharing construct node 4, one explicit barrier node 5, two implicit barriers nodes 7 and 9 and a loop (composed of nodes 4, 5, 6, 7). First Algorithm 1 removes the loop backedge from node 7 to node 4. Then sequential orders are computed: $C_0 = \{5\}$, $C_1 = \{7\}$ and $C_2 = \{9\}$. A warning is issued for the conditional node 4 as $PDF^+(C_0) = \{4\}$. For the loop construct node 4, the iterated postdominance frontier is empty.

3.3 Inter-Procedural Analysis

This section describes the analysis for the whole application code. We assume the application is not using recursion, meaning the callgraph of the application has no cycle.

The method iterates through the callgraph, in reverse topological order. It starts with functions that do not call other functions in the code, then callers of these functions, and so forth. After the previous analysis of Algorithm 1, each function retains the minimal number of barriers executed by the team of threads entering the function (excluding the barriers executed by teams created inside the function), as well as the number of worksharing constructs executed by this same team. These numbers are denoted $n_{\texttt{barrier}}$ for the number of barriers, n_d for worksharing constructs (among `for`, `worksharing`, `sections`, `single`). They are obtained through a simple traversal of the OMPCFG of the function. When a function g is called from a function f, g is replaced by as many barriers and worksharing constructs as these values. For worksharing constructs, only the number of constructs matters for the analysis. Indeed, we verify each callee function with worksharing constructs are not depending on the control flow in caller functions. Then the analysis `Function_verification` is called on f. These steps are described

Algorithm 2 OpenMP Inter-Procedural Analysis

```
 1: function CODE_VERIFICATION($CG, \upsilon$)        ▷ $CG$: call graph        ▷ $\upsilon$: level of verbosity
 2:     Sort $CG$ in reverse topological order
 3:     for $f \in CG$ do
 4:         for $g$ a callee in $f$ do
 5:             Compute $n_d(g)$ for $d$ =barrier and worksharing constructs        ▷ $n_d$: minimal
    number of directives $d$ executed by entering threads
 6:             Replace $g$ in $f$ by $n_d(g)$ empty worksharing constructs and $n_{\texttt{barrier}}(g)$ barriers.
 7:         end for
 8:         Compute Function_verification($f, \upsilon$)
 9:     end for
10: end function
```

in Algorithm 2.

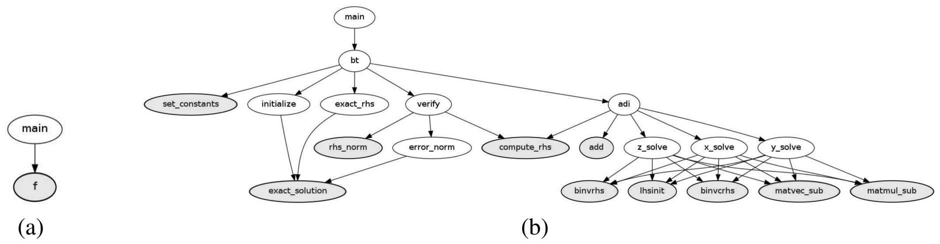


Fig. 4. Callgraph of Listing 1.2 (a) and BT from NASPB-OMP (b)

Figure 4 shows callgraphs of Listing 1.2 and BT from the NAS parallel benchmarks OpenMP. Nodes colored in gray are first nodes considered by Algorithm 2. In the example of the Listing 1.2 callgraph, Algorithm 2 computes $n_{worksharing}(f) = 0$ and

$n_{barrier}(f) = 0$ which are the minimal numbers of barriers and worksharing constructs in function `f`. Function f is then replaced by these numbers in `main`.

4 Experimental Results

Our analysis is implemented in a GCC 4.7.0 plugin, avoiding the whole compiler recompilation. An adaptation of the plugin is required to work with newer version of GCC. The plugin is located in the middle of the compilation chain as a new pass inserted inside the compiler pass manager after generating CFG informations and before OpenMP directives transformation. The location has the advantage of being language independent allowing a verification of applications written in C, C++ and Fortran. The pass applies Algorithm 1. The implementation of Algorithm 2 is currently under development. This section presents experimental results on the NAS parallel Benchmarks OpenMP (NASPB-OMP) [15] v3.2 using class B and HERA [3], a large multi-physics 2D/3D AMR hydrocode platform. Even if the test case used by HERA is parallelized with MPI+OpenMP, only the correctness of OpenMP barriers and worksharing constructs have been checked. The number of lines and the language of each benchmark is presented Table 2. All experiments were conducted on Tera 100, a supercomputer with a peak performance of 1.2 PetaFlops. Tera 100 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 were computed and averaged with BullxMPI 1.1.16.5.

Our analysis issues warnings for barriers and worksharing constructs potentially not encountered by all threads of a team. The name of the OpenMP directive with potential improper use and the line of the conditional leading to this situation are returned to the programmer. The following example shows what a user can read on `stderr` when compiling Listing 1.1 with our plugin.

```
in function 'f':
example.c: warning: STATIC-CHECK: #pragma omp single line 9 is
possibly not called by all threads because of the condition line 6
```

Table 2 shows the number of barriers and worksharing constructs found in each benchmark and the number of nodes in the sets S and S' generated by Algorithm 1 with the two verbosity levels. For all these nodes, the control flow does not depend on thread ID and therefore functions are correct. A data-flow analysis could be done to complement our analysis to reduce the number of false positives. Indeed, a check on the conditionals in $S \cup S'$ could help the plugin to detect control flow not depending on threads ID and avoid false positives. This is left for future work. The table also presents first results for the inter-procedural analysis by giving the number of functions executed in parallel with a non null minimal number of barriers or worksharing constructs. These functions may be replaced by their callee functions in the source code to report errors considering the entire program.

The compile-time overhead obtained when compiling the applications and activating our plugin is shown Figure 5. The overhead remains acceptable as it does not exceed 0.25% for NASPB-OMP and 10% for HERA (caused by the size of the code, it takes 52,3 minutes to compile HERA with the plugin).

Table 2. Static Results for each benchmark (F=FORTRAN)

Benchmark	NASPB-OMP											HERA
	BT	CG	DC	EP	FT	IS	LU	LU-HP	MG	SP	UA	
Language	F	F	C	F	F	C	F	F	F	F	F	C++
# lines	3,835	1,204	3,295	294	1,336	940	3,921	3,875	1,497	3,309	8,375	827,739
# explicit barriers	0	0	0	0	0	2	3	0	0	0	0	92
# worksharing	31	18	0	1	8	5	37	37	15	35	77	1,622
# nodes in $\mathbf{S} \cup \mathbf{S'}$	**Verbosity 0**											
	0	0	0	0	0	0	0	0	0	0	0	564
	Verbosity 1											
	0	0	0	0	0	0	0	0	0	0	0	587
# functions with $(n_{barrier} + n_d) \neq 0$	3	3	0	0	0	0	8	4	2	3	15	398

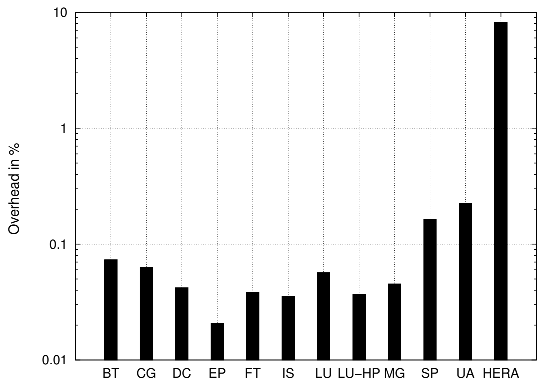


Fig. 5. Overhead of average compilation time for NASPB-OMP and HERA

5 Conclusion and Future Work

In this paper we propose an adaptation of our previous work on MPI to detect improper uses of barriers and worksharing constructs in OpenMP applications. The method we propose statically detects if all threads entering a function and created in it have the same sequence of barriers (or the same number of barriers) and worksharing constructs. It issues warnings for the statements responsible for the execution path leading to possible deadlocks or unspecified behaviors. Compared to existing work, in particular the method of Zhang *et al.* [18], our technique is fast (introducing little overhead) and able to scale to large applications. For future work, we plan to complement our method by a data-flow analysis, reducing the number of false positives detected by our approach.

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. Basupalli, V., Yuki, T., Rajopadhye, S., Morvan, A., Derrien, S., Quinton, P., Wonnacott, D.: ompVerify: Polyhedral Analysis for the OpenMP Programmer. In: Proceedings of the 7th International Conference on OpenMP in the Petascale Era. pp. 37–53. IWOMP'11 (2011)
2. Cytron, R., Ferrante, J., Rosen, B., Wegman, M., Zadeck, F.: Efficiently computing static single assignment form and the control dependence graph. In: ACM TOPLAS. pp. 13(4):451–490 (1991)
3. Jourdren, H.: HERA: A hydrodynamic AMR Platform for Multi-Physics Simulations. In: Plewa, T., Linde, T., Weirs, V.G. (eds.) Adaptive Mesh Refinement - Theory and Applications. pp. 283–294. Springer (2003)
4. Kim, Y.J., Daeyoung, K., Jun, Y.K.: An Empirical Analysis of Intel Thread Checker for Detecting Races in OpenMP Programs. In: Lee, R.Y. (ed.) ACIS-ICIS. pp. 409–414. IEEE Computer Society (2008)
5. Kim, Y.J., Park, M.Y., Park, S.H., Jun, Y.K.: A Practical Tool for Detecting Races in OpenMP Programs. In: Malyshkin, V.E. (ed.) PaCT. LNCS, vol. 3606, pp. 321–330. Springer (2005)
6. Kim, Y.J., Song, S., Jun, Y.K.: ADAT: An Adaptable Dynamic Analysis Tool for Race Detection in OpenMP Programs. In: ISPA. pp. 304–310. IEEE (2011)
7. Li, J., Hei, D., Yan, L.: Correctness Analysis based on Testing and Checking for OpenMP Programs. Fourth ChinaGrid Annual Conference, IEEE (2009)
8. Lin, Y.: Static Nonconcurrency Analysis of OpenMP Programs. In: Mller, M.S., Chapman, B.M., de Supinski, B.R., Malony, A.D., Voss, M. (eds.) IWOMP. LNCS, vol. 4315, pp. 36–50. Springer (2005)
9. Ma, H., Diersen, S., Wang, L., Liao, C., Quinlan, D.J., Yang, Z.: Symbolic Analysis of Concurrency Errors in OpenMP Programs. In: ICPP. pp. 510–516. IEEE (2013)
10. Meng, Y., Ha, O.K., Jun, Y.K.: Dynamic Instrumentation for Nested Fork-join Parallelism in OpenMP Programs. In: Proceedings of the 4th International Conference on Future Generation Information Technology. pp. 154–158. FGIT'12, Springer-Verlag (2012)
11. Petersen, P., Shah, S.: OpenMP Support in the Intel Thread Checker. In: Voss, M. (ed.) WOMPAT. LNCS, vol. 2716, pp. 1–12. Springer (2003)
12. Saillard, E., Carribault, P., Barthou, D.: Combining Static and Dynamic Validation of MPI Collective Communications. In: EuroMPI '13. pp. 117–122 (2013)
13. Terboven, C.: Comparing Intel Thread Checker and Sun Thread Analyzer. In: Bischof, C.H., Bcker, H.M., Gibbon, P., Joubert, G.R., Lippert, T., Mohr, B., Peters, F.J. (eds.) PARCO. Advances in Parallel Computing, vol. 15, pp. 669–676. IOS Press (2007)
14. GOMP site, gcc.gnu.org/projects/gomp
15. NASPB site, http://www.nas.nasa.gov/software/NPB
16. OpenMP API v4.0, http://www.openmp.org/
17. Yices: An SMT solver, http://yices.csl.sri.com
18. Zhang, Y., Duesterwald, E., Gao, G.R.: Concurrency analysis for shared memory programs with textually unaligned barriers. In: Adve, V.S., Garzarn, M.J., Petersen, P. (eds.) LCPC. LNCS, vol. 5234, pp. 95–109. Springer (2007)