MPI Thread-Level Checking for MPI+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. MPI is the most widely used parallel programming model. But the reducing amount of memory per compute core tends to push MPI to be mixed with shared-memory approaches like OpenMP. In such cases, the interoperability of those two models is challenging. The MPI 2.0 standard defines the so-called thread level to indicate how MPI will interact with threads. But even if hybrid programs are more common, there is still a lack in debugging tools and more precisely in thread level compliance. To fill this gap, we propose a static analysis to verify the thread-level required by an application. This work extends PARCOACH, a GCC plugin focused on the detection of MPI collective errors in MPI and MPI+OpenMP programs. We validated our analysis on computational benchmarks and applications and measured a low overhead.

Keywords: Static verification, OpenMP, MPI, MPI Thread level

1 Introduction

To address the challenges of exascale systems, MPI evolves to be mixed with shared-memory approaches like OpenMP. E. Lusk and A. Chan report for instance some successful use cases of OpenMP threads exploiting multiple cores per node with MPI communicating among the nodes [11]. But combining models does not facilitate the debugging task and requires special care for MPI calls [4]. Indeed, in an MPI+OpenMP program, not only the correctness of MPI should be ensured but also the multi-threaded model should not interfere with MPI. As an example, within a process, the same communicator may not be concurrently used by two different MPI collective calls. This means MPI collective operations may not be called by multiple parallel threads. The MPI-2 standard defines four thread-safety levels to indicate how MPI should interact with threads. According to the MPI standard, *it is the user responsibility to prevent races when threads within the same application post conflicting communication calls* ([17], p. 482). This should be checked above all for the fully multithreaded case (`MPI_THREAD_MULTIPLE`). This paper presents a static analysis to verify MPI Thread-level compliance required by an MPI+OpenMP application.

Listing 1.1.

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

Listing 1.2.

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

Listing 1.3.

```
void f(){
 /***/
 if(...){
  #pragma omp parallel
  {
   /***/
   #pragma omp master
   {
    MPI_Recv(..)
    MPI_Send(..)
   }
  }
 }
 /***/
}
```

Fig. 1. MPI+OpenMP examples showing different uses of MPI calls.

Fig. 1 illustrates some of the possible issues related to MPI communications in a multithreaded context through three examples. `MPI_Allreduce` in Listing 1.1 is called in a `single` block, `MPI_THREAD_SERIALIZED` then corresponds to the minimum level of compliance. However if the function `f` is called itself in a parallel construct, the collective is then executed in a nested parallel region, possibly leading to more than one concurrent call to this collective. This erroneous situation always occurs unless only one thread is created in the first parallel region or in both regions. Listing 1.2 illustrates a more complex case: two `MPI_Reduce`s are executed in `single` constructs in the same OpenMP parallel region. As the first construct contains a `nowait` clause, both `MPI_Reduce` can be executed concurrently by different threads. This requires a thread-level equal to `MPI_THREAD_MULTIPLE`, assuming the communicators used by the two collectives are different. If they are identical, the code is incorrect. In Listing 1.3, function `f` is compliant with the `MPI_THREAD_FUNNELED` level. However, if the master directive is replaced by a single directive, the `MPI_THREAD_SERIALIZED` level is the minimum thread-level required. Thus, these examples illustrate the difficulty for a developer to ensure that MPI calls are correctly placed inside an hybrid MPI+OpenMP application whatever the required thread-level support.

This paper proposes a static analysis that helps the application developer to check which thread-level support is required for a specific code. For this purpose, we suppose the programs are SPMD (Single Program Multiple Data) MPI programs. It means that every MPI rank calls the same functions in the same order. This covers a large amount of scientific simulation applications for High-Performance Computing. We integrated our analysis in the GCC plugin PARCOACH [13, 14] and we designed it to be compatible with other dynamic tools. Our paper makes the following contributions:

– Analysis to check the conformance of MPI+OpenMP codes with any MPI thread level (including `MPI_THREAD_MULTIPLE` level) defined in the MPI-2 standard and code transformation to verify the non-compliance at runtime.
– Full implementation inside a production compiler (GCC).

– Experimental results on multiple benchmarks and production applications.
– Functional integration with existing dynamic debugging tools (our approach is designed to be complementary to existing dynamic PMPI-based debugging tools like MUST[6])

This paper is organized as follows: Section 2 summarizes the related work on debugging of MPI and hybrid MPI+OpenMP applications, focusing on MPI thread-level compliance. Section 3 describes the basis of our approach. Then Section 4 exposes our static analysis detecting the thread-level compliance. Section 5 illustrates our approach on experimental results and finally Section 6 concludes.

2 Related Work

As most HPC applications are parallelized with MPI, a lot of work has been done to help programmers to debug MPI applications (TASS[15], DAMPI[21], MPI-CHECK[10], Intel Message Checker[2], Marmot[9], Umpire[20], MUST[6], MPICH[3]). Existing tools, static or dynamic, are able to detect the line in the source code where an error occured but rarely the line responsible for this situation. Although the compile-time offers the possibility to detect and correct possible errors earlier than at runtime, few tools rely on purely static analysis because of the combinatory aspect of methods used. We have developed in previous work a GCC plugin named PARCOACH[13, 14], to statically detect MPI collective errors in MPI and MPI+OpenMP programs. It combines compile-time code analysis with an instrumentation to prevent the application from deadlocking. This approach avoids systematic instrumentation, highlights conditionals that can lead to a deadlock and issues warnings with precise information.

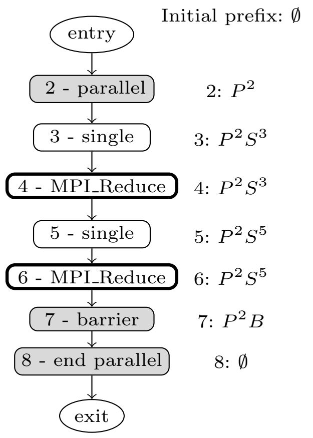
One of the MPI challenges is its interoperability with other programming models. Even if it is now possible to profile and visualize profiles and traces for MPI+OpenMP programs, debugging tools especially those detecting thread levels compliance are practically non-existent. To our knowledge, Marmot [5] is the only tool that provides a support for detecting violations in MPI+OpenMP programs. Marmot uses the MPI profiling interface (PMPI) to introduce artificial data races only occuring when some constraints are violated and detect them with the Intel Thread Checker tool. The authors define five restrictions for hybrid MPI applications based on the definition of the thread levels mentioned in the MPI standard. The fifth restriction is the non-violation to the provided thread level. However, as Marmot only relies on profiling, it may find for one run that the program is non compliant to a given thread level, and for another run find its compliance (so defining a compliance per run). The same happens for bugs, where detection may require many runs in a profile-only approach. On the contrary, PARCOACH finds statically the possible non-compliance of the code, pinpointing non-compliant code fragments and situations. The runtime instrumentation only checks whether these situations occur.

3 Analysis of the Multithreaded Context

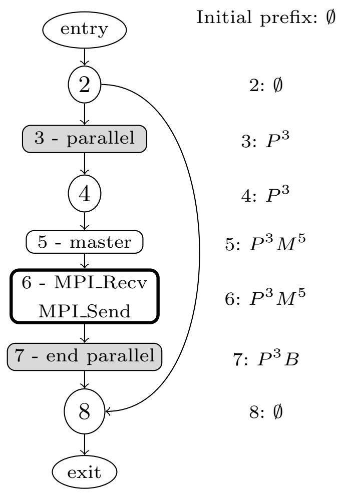
Our static analysis verifies the thread-level compliance of hybrid applications. The analysis proposed does not depend on one particular run and finds all possible situations of non-compliance to a given thread level. As it is conservative, it can be complemented by an instrumentation phase that checks the occurence of these situations. An essential part of the static analysis consist in determining the multithreaded context in which MPI calls (Point-to-point and collectives) are performed. The method described in this section computes a parallelism word to characterize this context in each point of the function analyzed.

3.1 Parallelism Words Construction

The analysis operates on the code represented as an intermediate-code form. We consider the program is represented as a *control-flow graph* (CFG), built in almost all compilers. The compile-time verification then consists in a static analysis of the CFG for each function of a program. The CFG is defined as a directed graph with artificial entry and exit nodes. Each node corresponds to a *basic block* and has a set of successors and predecessors. The CFG is augmented to highlight nodes containing MPI calls (collectives and P2P). As for the GCC compiler, OpenMP directives are put into separate basic blocks. Hence new nodes are added for explicit and implicit thread barriers. For sake of clarity, implicit thread barriers at the end of parallel regions are denoted by `end parallel`.



(a) CFG of Listing 1.2



(b) CFG of Listing 1.3

Fig. 2. Control Flow Graph and parallelism words of Listings 1.2 and 1.3

To highlight the thread context in which an MPI call is performed, we extend the notion of parallelism words defined in [14], taking into account the needs of a thread level compliance analysis. The *parallelism word* of a basic block is the sequence of OpenMP parallel constructs (pragma `parallel`, `single`, ...) surrounding this block and the barriers traversed from the beginning of a function to the block. Parallel regions containing the block are denoted by P^i, with i the id of the basic block with the OpenMP construct. Similarly, regions executed by the master thread are denoted by M^i and other single threaded regions are denoted S^i. Finally, `barrier` corresponds to B. OpenMP defines a perfectly-nested parallelism, thus the control flow has no impact on the parallelism word. Each node (basic block) n is associated to a parallelism word denoted $pw[n]$. With a depth-first search starting at the entry node, each node then sets its parallelism word depending on its predecessor and the OpenMP directives it contains. P is added when a parallel region is encountered, S is added when a single, section or task region is traversed, M is added when a master construct is traversed and B is added when an implicit or explicit thread barrier is met. Figure 2 shows examples of CFG with their associated parallelism words.

3.2 Parallelism Words Analysis

The automaton Figure 3 defines the possible parallelism words. Nestings forbidden by the OpenMP specification (SS, MS,...) are not considered by the automaton. If the target obtains such forbidden nested regions, our analysis returns the error message: *invalid state, error*. The language of accepted parallelism words will depend on the specified thread level. As we check each function independently, the level of parallelism in which a function is called is unknown. To provide an accurate picture of the level of thread parallelism in which function occurrence is called, the statistics on the NAS Parallel Benchmarks multizone (NASPB-MZ) using class B [18] have been collected and are shown in Table 1 per thread, in each process. We notice that functions are mainly called within one level of multithreading. Thus to consider all possible initial conditions, each callsite is instrumented in order to capture the initial parallelism word of each function. This word corresponds to a prefix P_i for all basic blocks of the called function and defines an initial state in Automaton Figure 3 (all states are possible initial states). The user can choose the initial state at compile-time.

Table 1. Level of threads parallelism at function entries for NASPB-MZ

Benchmark	# function calls	# calls in state 0,2,3	# calls in state 1,4	# calls in state 5,6
BT-MZ	396,918,403	45,379	396,873,024	0
SP-MZ	15,479,425	116,161	15,363,264	0
LU-MZ	3,017,513	40,745	2,976,768	0

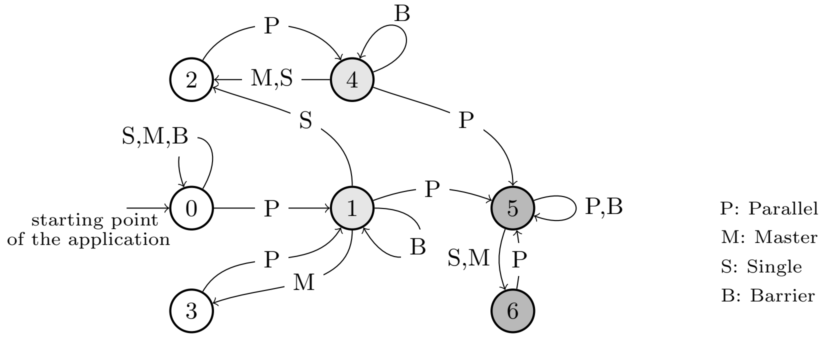


Fig. 3. Automaton of possible parallelism words. Nodes 0, 2 and 3 correspond to code executed by the master thread or a single thread. Nodes 1 and 4 correspond to code executed in a parallel region, and 5 and 6 to code executed in nested parallel region.

The following section describes the analysis checking the thread-level compliance based on the parallelism words of the basic blocks containing MPI communications.

4 Thread-Level Compliance Checking

This section describes how the non-compliance of thread levels can be detected at compile-time. For that purpose we use parallelism words introduced in the previous section to check the placement of MPI calls within a process.

4.1 Static Analysis and Interface to Dynamic Checkings

For each possible thread level we define a language of valid parallelism words based on the automaton Fig. 3. For a given basic block, its parallelism word consists in the prefix (obtained from the callsite of the function or user-defined) and the word computed from previous analysis. The analysis verifies if nodes containing MPI calls (P2P and collectives) are associated with an accepted word. Thread barriers can be safely ignored as they do not influence the level of thread parallelism. In case of the detection of a possible error, a warning related to the initial level with the name of the call is returned to the programmer. Alg. 1 takes as input the CFG and the language L of correct parallelism words and outputs the sets S and S_{ipw}. These sets respectively contain the nodes violating the input language and the nodes that dominate these nodes before the execution/control flow changes. This set will be given as one of the input parameters of the dynamic analysis. In the algorithm, line 5, the node u corresponds to the node preceeding n in the CFG and that is the immediate successor of a control flow node (with two successors) or of a pragma node (changing the parallelism word). The nodes in the set S_{ipw} correspond to execution points where compliance should be tested at runtime, in order to handle possible false-positives detected statically. A unique parallelism word is computed at runtime and updated after each OpenMP construct. Compared to the compile-time parallelism words, parallel regions created with only one thread correspond to the parallelism word ϵ. This implies that such region has no impact on the current multithreaded context. The insertion of such computations and checks can be conducted in tools such as MUST [6], Marmot [9] or following the techniques proposed in [12].

Algorithm 1 Detection of *parallelism words* for multithreaded regions

```
 1: function MULTITHREADED_REGIONS($G = (V, E), L$)        ▷ $G$: CFG, $L$: language
 2:     $S_{ipw} \leftarrow \emptyset$ , $S \leftarrow \emptyset$
 3:     for each $n \in V | n$ contains a MPI call do
 4:         if $pw[n] \notin L$ then
 5:             $u \leftarrow$ Node that dominates $n$ before execution/control flow changement
 6:             $S \leftarrow S \cup \{n\}$, $S_{ipw} \leftarrow S_{ipw} \cup u$
 7:         end if
 8:     end for
 9:     Output nodes in $S$ as warnings
10: end function
```

4.2 MPI_THREAD_SINGLE

By setting the MPI_THREAD_SINGLE level, the user ensures only one thread will execute MPI calls ([17], p. 486). This means all MPI calls should be performed outside multi-threaded regions. Thus all nodes of the CFG containing a MPI call must be associated with an empty parallelism word. The language L of accepted parallelism words is then defined by $L = \{\epsilon\}$. Algorithm 1 with $L = \{\epsilon\}$ returns the non-compliant MPI calls (set S).

4.3 MPI_THREAD_FUNNELED

The use of MPI_THREAD_FUNNELED level means the process may be multithreaded but the application must ensure that only the thread that initialized MPI can make MPI calls ([17], p. 486). For this level, State 3 in Automaton 3 is the accepting state and the language $L = (PB^*M)^+$ describes the accepted words. With Algorithm 1 and L, our analysis detects MPI calls that are not executed in a master region.

4.4 MPI_THREAD_SERIALIZED

The MPI_THREAD_SERIALIZED level means the process may be multi-threaded but only one thread at a time can perform MPI calls ([17]). The accepting states in Automaton 3 are states 2 and 3. Thus, the language $L = (PB^*S|PB^*M)^*$

describes the accepted words. This language contains parallelism words ending by S or M without a repeated sequence of P. Critical sections and locks are not supported here and is part of our future work.

To verify the compliance of this level, Algorithm 1 is used to make sure all MPI calls are performed in a monothreaded context. Different MPI calls in the same monothreaded region are sequentially performed as only one thread executes it. However, calls in different monothreaded regions may be called simultaneously if monothreaded regions are executed in parallel (no thread synchronization between monothreaded regions). Special care is requested for MPI collective operations. All MPI processes should execute the same sequence of MPI collective operations in a deterministic way. That means there is a total order between MPI collective calls. Algorithm 2 shows the detection of concurrent calls. It takes as input the CFG and outputs two sets: S and S_{cc}. When nodes containing a MPI call with the same number of B are detected these nodes are put in the set S and the nodes that begin the monothreaded regions are put in the set S_{cc} for the dynamic analysis. A warning is issued for nodes in S.

Algorithm 2 Detection of potential concurrent calls

```
1: function CONCURRENT_CALLS($G = (V, E)$)                    ▷ $G$: CFG
2:     $S_{cc} \leftarrow \emptyset$, $S \leftarrow \emptyset$
3:     Remove loop back edges
4:     if $\exists\ u, v \in$ nodes in concurrent monothreaded regions then
5:         $i, j \leftarrow$ nodes immediate successors of nodes creating monothreaded regions
6:         $S \leftarrow S \cup \{u, v\}$, $S_{cc} \leftarrow S_{cc} \cup \{i, j\}$
7:     end if
8:     Output nodes in $S$ as warnings
9: end function
```

To dynamically verify the total order of MPI collective sequences in each MPI process, validation functions are inserted in nodes in the sets S_{ipw} and S_{cc} generated by Algorithms 1 and 2: CC_{ipw} and CC_{cc}. Function CC_{ipw} detects incorrect execution parallelism words and Function CC_{cc} detects concurrent collective calls. In Figure 2, nodes 4 and 6 have the same number of thread barriers in their parallelism words (node 4: P^2S^3, node 6: P^2S^5) so the collective operations involved are potential concurrent collective calls. Indeed, the `nowait` clause remove the implicit barrier at the end of the first single region. The algorithm outputs a warning for collective calls located nodes 4 and 6 ($S = \{4, 6\}$) and flags nodes 4 and 6 for dynamic checks ($S_{cc} = \{4, 6\}$). CC_{cc} functions are then inserted in nodes 4 and 6.

4.5 MPI_THREAD_MULTIPLE

This level is the least restrictive level. It enables multiple threads to call MPI with no restriction ([17], p. 486). However MPI calls should be thread safe, meaning that when two concurrently running threads make MPI calls, the outcome will be as if the calls executed sequentially in some order. The verification of this level follows the same analyses as for the MPI_THREAD_SERIALIZED level.

5 Experimental results

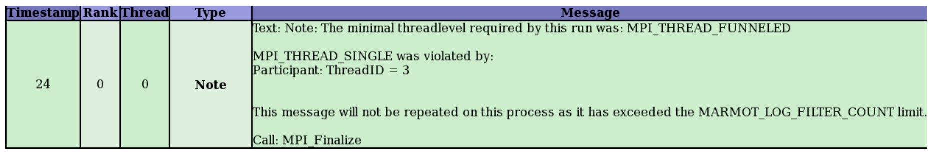
This section is intended to show the impact of our analysis on the compilation time. For that purpose we present experimental results obtained on the NAS Parallel benchmarks multizone (NAS-MZ v3.2) using class B [18], five MPI+OpenMP Coral benchmarks [19] (AMG2013, LULESH, HACC, SNAP, miniFE) and a production test case named HERA [8], which is a large multi-physics 2D/3D AMR hydrocode platform. To highlight the functionality of our analysis, we created a microbenchmark suite called BenchError containing five hybrid programs that violate thread level constraints (coll_single, coll_funneled, coll_serialized, p2p_multiple) and contain MPI collective (coll_deadlock) errors. All compilation experiments were conducted on the Tera-100 supercomputer (peak performance of 1.2 PFlops) and computed with BullxMPI 1.1.16.5.

5.1 Functionnalities of the Analysis

We extended PARCOACH, a GCC plugin located in the middle end of the compilation chain after the CFG generation and before OpenMP directives transformation. Hence the plugin is language independent allowing the verification of programs written in C, C++ and Fortran. Our analysis is therefore simple to deploy in existing environment as it does not modify the whole compilation chain. The analysis issues warnings at compile-time with potential error information (lines of MPI calls, line where the dynamic check is inserted,...). The following example shows what a user can read on *stderr* when compiling the program coll_serialized corresponding to Listing 1.2.

```
in function 'f':
Warning: PARCOACH: possible non-compliance of MPI_THREAD_SERIALIZED level. Potential concurrent
coll. calls within a process : MPI_Reduce l.11 may be called simultaneously with MPI_Reduce l.6
PARCOACH: Minimum thread-level required: MPI_THREAD_MULTIPLE
PARCOACH inserted a check after the single directive l.4 | the single directive l.9
```

In this example, `MPI_Reduce` calls were done on different communicators. As our analysis does not check communicators, both single regions are instrumented to check if the non-compliance of the thread level is confirmed at runtime. In comparison, the error message returned by Marmot at runtime is the following:



Timestamp	Rank	Thread	Type	Message
24	0	0	**Note**	Text: Note: The minimal threadlevel required by this run was: MPI_THREAD_FUNNELED MPI_THREAD_SINGLE was violated by: Participant: ThreadID = 3 This message will not be repeated on this process as it has exceeded the MARMOT_LOG_FILTER_COUNT limit. Call: MPI_Finalize

Marmot finds that the code should be executed within the `MPI_THREAD_FUNNELED` thread level whereas PARCOACH finds the level `MPI_THREAD_MULTIPLE`. The reason comes from the fact that Marmot detects conformance w.r.t. one execution, and in particular to one parallel schedule. During the execution monitored by Marmot, the `single` constructs are executed by the master thread leading to a serialized sequence of these constructs. However, from a conformance point of view, this is incorrect and the thread level `MPI_THREAD_MULTIPLE` as analyzed by PARCOACH should be chosen.

5.2 Static Analysis Results

Table 2 shows the language and the number of lines of each benchmark we tested. The 4^{th} and 5^{th} columns depict the thread level provided (level actually returned to the user, might be lower than the desired level, depending on the MPI implementation) and the minimum thread level required by the application (thread-level the user should use). The last column displays the compliance our analysis returned. Our analysis was able to find the thread-level non-compliance in our microbenchmark suite. Notice that the `MPI_THREAD_MULTIPLE` level was not supported by the MPI implementation we used. For each benchmark, the overhead obtained at compile-time (serial compilation) is presented Figure 4. This overhead is acceptable as it does not exceed 6%.

Table 2. Compliance Results

Benchmark	Language	Lines of code	Thread level provided	Thread level required	Compliant
BT-MZ || SP-MZ	Fortran	6,779 || 4,862	`SINGLE`	`SINGLE`	yes
LU-MZ	Fortran	6,542	`SINGLE`	`SINGLE`	yes
AMG2013 || LULESH	C	75,000 || 5,000	`SINGLE`	`SINGLE`	yes
miniFE || HACC	C++	50,000 || 35,000	`SINGLE`	`SINGLE`	yes
SNAP	Fortran	3,000	`SINGLE`	`SINGLE`	yes
HERA	C++	500,000	`SERIALIZED`	`SERIALIZED`	yes
coll_single	C	29	`SINGLE`	`FUNNELED`	no
coll_funneled	C	36	`FUNNELED`	`SERIALIZED`	no
coll_serialized	C	47	`SERIALIZED`	`MULTIPLE`	no
coll_deadlock	C	38	`FUNNELED`	`FUNNELED`	yes
p2p_multiple	C	45	`SERIALIZED`	`MULTIPLE`	no

PARCOACH issues warnings for potential MPI collective errors within an MPI process and between processes. The type of each potential error is specified (collective mismatch, concurent calls in an MPI process,...) with the names and lines in the source code of MPI collective calls involved. Table 3 shows the number of static MPI collective calls and the number of nodes in the set S found by PARCOACH (algorithms 1 and 2 of our analysis). The 4th column depicts the percentage of the benchmarks functions instrumented. We notice a good impact of the static analysis on the selective instrumentation. The two last columns give the number of expected errors and the number of errors actually found.

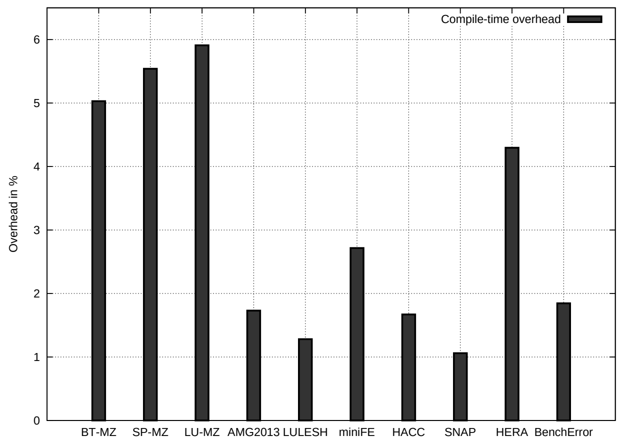


Fig. 4. Overhead of average compilation time

Table 3. Debugging Results

Benchmark	# collective calls	# nodes in S	% instrumented functions	# expected errors	# errors found
BT-MZ || SP-MZ	15 || 15	7 || 7	8,57% || 8,57%	0 || 0	0 || 0
LU-MZ	20	7	8,82%	0	0
AMG2013	86	75	13.33%	0	0
LULESH || miniFE	3 || 4	1 || 6	1.44% || 2.56%	0 || 0	0 || 0
HACC || SNAP	26 || 9	11 || 13	1.41% || 10%	0 || 0	0 || 0
HERA	574	375	<1%	0	0
coll_single || coll_funneled	1 || 1	1 || 1	100% || 100%	1 || 1	1 || 1
coll_serialized	2	2	100%	1	1
coll_deadlock	1	1	100%	1	1
p2p_multiple	0	2	100%	1	1

6 Conclusion and Future Work

Augmenting MPI applications with OpenMP constructs is one possible approach to face exascale systems. But the development of such hybrid applications requires effective debugging methods to assist programers. In this paper, we presented a compiler analysis to verify the MPI thread-level compliance of C/C++ and Fortran MPI+OpenMP codes. The analysis proposed finds the right MPI thread level to be used and identifies code fragments that may prevent conformance to a given level. We have shown a small impact on compilation-time with an overhead lower than 6%. For future work, our analysis could be extended to include critical sections and locks. Furthermore, it could be integrated into existing tools like Marmot or MUST to cover other errors like calls arguments (e.g., communicators) or to report warnings concerning the execution path responsible for bugs related to thread-level MPI compliance.

References

1. Chiang, W.F., Szubzda, G., Gopalakrishnan, G., Thakur, R.: Dynamic Verification of Hybrid Programs. pp. 298–301. EuroMPI, Springer-Verlag (2010)
2. DeSouza, J., Kuhn, B., de Supinski, B.R., Samofalov, V., Zheltov, S., Bratanov, S.: Automated, Scalable Debugging of MPI Programs with Intel Message Checker. In: SE-HPCS'05. pp. 78–82. ACM (2005)
3. Falzone, C., Chan, A., Lusk, E., Gropp, W.: A Portable Method for Finding User Errors in the Usage of MPI Collective Operations. IJHPCA 21(2), 155–165 (2007)
4. Gropp, W., Thakur, R.: Thread Safety in an MPI Implementation: Requirements and Analysis. Parallel Computing 33(9), 595–604 (2007)
5. Hilbrich, T., Müller, M.S., Krammer, B.: Detection of Violations to the MPI Standard in Hybrid OpenMP/MPI Applications. In: IWOMP. pp. 26–35 (2008)
6. 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. In: EuroMPI. pp. 129–134 (2013)
7. Hilbrich, T., Protze, J., de Supinski, B.R.d., Schulz, M., Müller, M.S., Nagel, W.E.: Intralayer Communication for Tree-Based Overlay Networks. In: Intl. Conf. on Parallel Processing. pp. 995–1003 (2013)
8. 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)
9. Krammer, B., Bidmon, K., Müller, M.S., Resch, M.M.: MARMOT: An MPI Analysis and Checking Tool. In: PARCO. Advances in Parallel Computing, vol. 13, pp. 493–500. Elsevier (2003)
10. Luecke, G.R., Chen, H., Coyle, J., Hoekstra, J., Kraeva, M., Zou, Y.: MPI-CHECK: a tool for checking Fortran 90 MPI programs. Concurrency and Computation: Practice and Experience 15(2), 93–100 (2003)
11. Lusk, E., Chan, A.: Early Experiments with the OpenMP/MPI Hybrid Programming Model. In: IWOMP. pp. 36–47. Springer-Verlag (2008)
12. Saillard, E., Carribault, P., Barthou, D.: Combining static and dynamic validation of MPI collective communications. pp. 117–122. EuroMPI, ACM (2013)
13. Saillard, E., Carribault, P., Barthou, D.: PARCOACH:Combining Static and Dynamic Validation of MPI Collective Communications. IJHPCA (2014)
14. Saillard, E., Carribault, P., Barthou, D.: Static/Dynamic Validation of MPI Collective Communications in Multi-threaded Context. In: PPoPP. ACM (2015)
15. Siegel, S., Zirkel, T.: Automatic Formal Verification of MPI Based Parallel Programs. In: PPoPP. pp. 309–310 (2011)
16. Smith, L., Bull, M.: Development of Mixed Mode MPI/OpenMP Applications. Sci. Program. 9(2,3), 83–98 (2001)
17. Message Passing Interface Forum. http://www.mpi-forum.org/docs/docs.html
18. NASPB site: http://www.nas.nasa.gov/software/NPB
19. CORAL site: https://asc.llnl.gov/CORAL-benchmarks/
20. Vetter, J.S., de Supinski, B.R.: Dynamic Software Testing of MPI Applications with Umpire. In: ACM/IEEE Conf. on Supercomputing (2000)
21. Vo, A., Aananthakrishnan, S., Gopalakrishnan, G., de Supinski, B.R.d., Schulz, M., Bronevetsky, G.: A Scalable and Distributed Dynamic Formal Verifier for MPI Programs. In: ACM/IEEE SC10. pp. 1–10 (2010)
22. Wolff, M., Jaouen, S., Jourdren, H.: High-order dimensionally split lagrange-remap schemes for ideal magnetohydrodynamics. In: Discrete and Continuous Dynamical Systems Series S. NMCF (2009)