# Static/Dynamic Validation of MPI Collective Communications in Multi-threaded Context

Emmanuelle Saillard
CEA, DAM, DIF, F-91297 Arpajon, France
emmanuelle.saillard@cea.fr

Patrick Carribault
CEA, DAM, DIF, F-91297 Arpajon, France
patrick.carribault@cea.fr

Denis Barthou
Bordeaux Institute of Technology / LaBRI INRIA, Bordeaux, France
denis.barthou@labri.fr

## Abstract

Scientific applications mainly rely on the MPI parallel programming model to reach high performance on supercomputers. The advent of manycore architectures (larger number of cores and lower amount of memory per core) leads to mix MPI with a thread-based model like OpenMP. But integrating two different programming models inside the same application can be tricky and generate complex bugs. Thus, the correctness of hybrid programs requires a special care regarding MPI calls location. For example, identical MPI collective operations cannot be performed by multiple non-synchronized threads. To tackle this issue, this paper proposes a static analysis and a reduced dynamic instrumentation to detect bugs related to misuse of MPI collective operations inside or outside threaded regions. This work extends PARCOACH designed for MPI-only applications and keeps the compatibility with these algorithms. We validated our method on multiple hybrid benchmarks and applications with a low overhead.

***Categories and Subject Descriptors*** D.2.5 [*Software Engineering*]: Testing and Debugging aids

***Keywords*** Static, Verification, MPI+OpenMP, Control Flow

## 1. Introduction

Hybrid programming with MPI and a shared-memory programming language such as OpenMP is a promising solution for writing parallel applications for supercomputers. The MPI-2 standard defines multiple levels of multi-threading integration, known as *thread levels*, to indicate how MPI communications should interact with threads. Even if some hybrid applications do not require a specific thread level, performing MPI communications inside parallel threaded regions may help reducing the penalty of Amdhal's law. Thus, three thread levels enable MPI communications inside OpenMP parallel regions. However, according to the standard, it is the user's responsibility to ensure that MPI communications (including collective calls) are correctly placed, according to the thread level used. More specifically, if the number of expected calls to a collective operation or their sequence is not the same for all processes, this can lead to errors or deadlocks. Finding such bugs and, moreover, the source of the errors may be challenging.

Even if hybrid applications are more and more common, most debugging tools are focused on one type of parallelism at a time. However errors in hybrid programs (whatever the thread-level support) can result from the combination of both forms of parallelism. To our knowledge, Marmot [1] is the only tool that provides a support for detecting collective errors in MPI+OpenMP programs.

We propose an extension of PARCOACH [4] to verify the flow of MPI collective operations in a multi-threaded context. The proposed method is compatible with all possible thread levels. It detects deadlocks or error situations due to MPI collectives, stops program execution as soon as this situation is unavoidable and reports to the user the control-flow divergence and the parallel constructs responsible for this situation. Our analysis is designed to be compatible with existing dynamic tools like MUST [2] and is focused on detecting the MPI collective mismatches in a multi-threaded context. The correctness of collectives arguments or the multi-threaded model used is not checked. The multi-threaded model should be an explicit fork/join model, with perfectly nested regions. OpenMP corresponds to this kind of model. Throughout the rest of the paper we consider MPI+OpenMP programs.

## 2. Compile-Time Verification

In our context, the problem statement can be expressed as follows: A hybrid program is correct if all MPI processes execute the same MPI collective operations in the same order in a deterministic way. This means there is a total order between MPI collective calls within each process and this order is the same for all MPI processes. To prove that a hybrid program is correct, the analysis is decomposed into three phases:

1. All MPI collectives are executed in a monothreaded context;
2. Any two collective executions are ordered sequentially;
3. All MPI processes execute the same sequence of collectives.

(1) and (2) ensure that collective operations are executed in an order that does not depend on the number of threads, nor on their execution schedule. (3) shows that the same sequence of collectives is executed for all MPI processes, and when the compile-time analysis is not able to prove this property due to some control flow statements, checks are inserted at these statements.

The compile-time phase of PARCOACH takes place in the middle of the compilation chain where the code is represented as a *control-flow graph* (CFG). In addition to the modification done in [4] to highlight nodes containing a MPI collective operation, OpenMP directives are put into separate basic blocks and new nodes are added for implicit thread barriers. To verify the total or-



*PPoPP'15*, February 7–11, 2015, San Francisco, CA, USA
ACM 978-1-4503-3205-7/15/02
http://dx.doi.org/10.1145/2688500.2688548

der of MPI collective calls, we define a *parallelism word* $pw[n]$ for a node $n$ as the sequence of the parallel constructs (pragma `parallel`, `single`, ...) and the barriers traversed from the begining of a function to the node. Parallel regions are denoted by $P^i$, with $i$ the id of the node with the OpenMP construct, single threaded regions (such as `single, master`,...) are denoted similarly $S^i$, and `barrier` by $B$. A simplification is done when OpenMP regions end. Because the considered thread-based models such as OpenMP have perfect nested parallelism, the control flow has no impact on the parallelism word.

Checking that a collective is executed in a monothreaded region boils down to check the parallelism word of its node. This requires to use MPI with at least the `MPI_THREAD_SERIALIZED` level. If a collective is executed in a multithreaded region, this requires further the use of the level `MPI_THREAD_MULTIPLE` and the code is then correct if only one thread executes the collective. To be in a monothreaded region, the parallelism word has to end with an $S$ ($B$s are ignored as barriers do not influence the level of thread parallelism). Moreover, if the parallelism word has a sequence of two or more $P$ with no $S$ in-between, it implies the parallelism is nested: Even if the word ends with an $S$, one thread for each thread team can execute the collective. We assume then the collective is not executed in a monothreaded region. The language $L$ defined by $L = (S|PB^*S)^*$ describes the accepted words. The initial *parallelism word* at the function entrance is considered as an empty word. A node $n$ is then in a monothreaded context if $pw[n] \in L$. As in practice, the initial *parallelism word* of the function is an initial prefix unknown at compile-time, the programmer can select with an option given to the analysis the initial level to consider at compile-time. Whenever a collective is in a multi-threaded context, a warning related to the initial level with the name of the collective is returned to the programmer. Two sets are created: $S$ and $S_{ipw}$ containing respectively collective nodes in multithreaded regions and the nodes that dominate these collective nodes before the execution/control flow changes.

Different MPI collectives can be called in monothreaded regions, and still be executed simultaneously if the regions are executed in parallel. The second step of the static analysis detects concurrent collective calls. Two nodes $n_1$ and $n_2$ are said to be in *concurrent monothreaded regions* if they are in monothreaded regions and if $pw[n_1] = wS^ju$ and $pw[n_2] = wS^kv$ where $w$ is a common prefix, $j \neq k$, $u$ and $v$ words in $(P|S|B)^*$. Two nodes in monothreaded regions can be executed simultaneously if and only if they are in concurrent monothreaded regions. Two sets are created: $S$ and $S_{cc}$. When collective nodes 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 $S_{cc}$.

Once the sequence of MPI collective calls is verified in each process we must check that all sequences are the same for all processes. To this end, we resort to the Algorithm 1 proposed in [4].

## 3. Static Instrumentation for Execution-Time Verification

The static analysis could lead to false positives relatively to the CFG that is possibly not correlated with the actual control flow. To deal with false positive results, a dynamic instrumentation is added in the nodes created by the static analysis ($S$, $S_{ipw}$ and $S_{cc}$).

For each node in $S_{ipw}$ and $S_{cc}$ a runtime check is done to ensure the node is actually in a monothreaded region. For each node in $S_{cc}$ the number of threads concurrently executing a given node is counted dynamically. The check function $CC$ depicted Algorithm 3 in [4] is inserted before each MPI collective operation and before `return` statements. As multiple threads may call $CC$ before `return` statements, this function is wrapped into a `single` pragma.

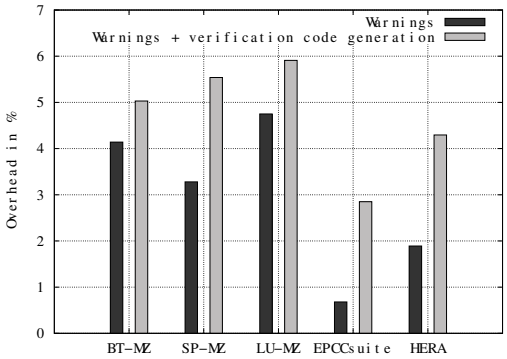


**Figure 1.** Overhead of average compilation time with and without verification code generation

In all cases if an error is about to occur, the program is stopped and an error message is returned with error type information.

## 4. Experimental Results

We tested our analyses on the NAS Parallel benchmarks multizone (NASPB-MZ v3.2) using class B, a mixed mode MPI/OpenMP benchmark suite v1.0 (EPCC suite) and HERA [3], a large multiphysics 2D/3D AMR hydrocode platform. At compile-time our analysis issues warnings for potential MPI collective errors within an MPI process and between MPI processes. The type of each potential error is specified (collective mismatch, concurent collective calls,...) with the names and lines in the source code of MPI collective calls involved. Figure 1 presents the compile-time overhead with and without code generation for each benchmark. The overhead acquired is acceptable as it does not exceed 6%.

## 5. Conclusion

Although large MPI+Threads applications appear, the lack of debugging tools for hybrid programs does not encourage the development of such applications and limits the use of thread levels. In this paper we propose a method to overcome this issue, extending PARCOACH to detect collective patterns that can raise errors/deadlocks in a multi-threaded context. First our method statically identifies MPI collective operations that can deadlock or be performed by multiple non-synchronized threads. Then we validate these potential errors/deadlocks during execution by a code transformation. The cost of the runtime checks is limited by a selective instrumentation, avoiding unnecessary checks.

## References


[1] T. Hilbrich, M. S. Müller, and B. Krammer. Detection of violations to the MPI standard in hybrid OpenMP/MPI applications. In *Intl. Conf. on OpenMP in a New Era of Parallelism*, pages 26–35, 2008.

[2] T. Hilbrich, B. R. de Supinski, F. Hänsel, M. S. Müller, M. Schulz, and W. E. Nagel. Runtime MPI collective checking with tree-based overlay networks. In *European MPI Users' Group Meeting*, pages 129–134, 2013.

[3] H. Jourden. HERA: A hydrodynamic AMR Platform for Multi-Physics Simulations. In T. Plewa, T. Linde, and V. G. Weirs, editors, *Adaptive Mesh Refinement - Theory and Applications*, pages 283–294, 2003.

[4] E. Saillard, P. Carribault, and D. Barthou. PARCOACH:combining static and dynamic validation of mpi collective communications. *Intl. Journal on High Performance Computing Applications (IJHPCA)*, 2014.