# Correctness Analysis of MPI-3 Non-Blocking Communications in PARCOACH

Julien Jaeger
CEA, DAM, DIF
F-91297 Arpajon, FRANCE
julien.jaeger@cea.fr

Emmanuelle Saillard
CEA, DAM, DIF
F-91297 Arpajon, FRANCE
emmanuelle.saillard.ocre@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@inria.fr

## ABSTRACT

MPI-3 provide functions for non-blocking collectives. To help programmers introduce non-blocking collectives to existing MPI programs, we improve the PARCOACH tool for checking correctness of MPI call sequences. These enhancements focus on correct call sequences of all flavor of collective calls, and on the presence of completion calls for all non-blocking communications. The evaluation shows an overhead under 10% of original compilation time.

## Keywords

MPI, Non-blocking collectives, checker, static analysis

## 1. INTRODUCTION

Most parallel scientific applications rely on the distributed-memory specification called MPI (Message Passing Interface) to efficiently exploit a supercomputer and reach high parallel performance. This programming model proposes point-to-point and collective communications to transfer messages between the different processes. Furthermore, the first specification provides the notion of non-blocking point-to-point communications.

**Non-Blocking Point-To-Point Communications and Completion Calls**: Non-blocking point-to-point communication allows to overlap communication and computation and thus to leverage hardware parallelism. Several studies [2] have shown that the performance of parallel applications can be significantly enhanced with overlapping techniques. A call to a non-blocking point-to-point communication initiates an action without completing it. Another function, called *completion call* is necessary to finish the action. At the end of this completion call, the runtime ensures that it is safe for the application to reuse the communication buffers. One can distinguish two types of completion calls: completion calls that complete only one non-blocking operation ( functions `MPI_{Wait,Waitany,Test,Testany}`, referred to as *unitary completion call* or *ucc*) and completion calls checking in a list of operations all the ones that are done (functions `MPI_{Waitall,Waitsome,Testall,Testsome}`, referred to as *multi completion call* or *mcc*).

**Collective Communications**: Collective communications are an important part of parallel scientific computing [1]. The non-blocking collectives provided in MPI-3 API combine the efficient algorithms of collectives operations with the overlapping benefits of non-blocking communications. The non-blocking collective model is similar to the one proposed by non-blocking point-to-point communications. The main difference with point-to-point communications is that blocking collectives can not be matched to their non-blocking counterparts. The ordering of collectives between blocking and non-blocking collectives, and between non-blocking collectives, is very important. If one process calls a non-blocking collective, the remaining processes should also call the same function before initiating any other collective.

In this paper, we introduce an evolution of the PARCOACH [3] tool with a new static analysis allowing to debug MPI non-blocking call sequences.

## 2. ANALYSIS OF NON-BLOCKING COMMUNICATIONS

To tackle the issues pointed out in Section 1, we propose two new analyses: (i) a new static pass to check if the number of completion calls may match non-blocking call (either collective or point-to-point) and (ii) a static analysis to check that the sequence of all flavor of collective is the same along all possible execution paths. These analyses have been implemented inside PARCOACH [3], a tool proposing a two-phase analysis to detect incorrect collective patterns in MPI programs. It combines a static analysis identifying the reduced set of problematic collectives with a selective instrumentation on the corresponding nodes in the Control Flow Graph (CFG) of the function to handle the deadlocks.



*EuroMPI '15 September 21-23, 2015, Bordeaux , France*



ACM ISBN 978-1-4503-3795-3/15/09.

DOI: http://dx.doi.org/10.1145/2802658.2802674

**Matching Non-blocking Communication with Completion Calls:** With this new pass, we determine if each non-blocking operation, either point-to-point or collective, can be matched to a completion call. For a path to be valid, each non-blocking operation should correspond to a *unitary completion call* or to a *multi completion call*. The exact matching would require to execute the code, checking all arguments passing and memory movements. Statically, we make the following assumption: a completion call always matches a previous pending non-blocking operation, if any. Within this assumption and for each basic block of the analyzed function, the number of pending non-blocking communications (called *pnb*) is computed, using an interval notation, following Algorithm 1. The lower bound (*lb*) corresponds to the number of pending communications when *mcc* statements complete all the previous non-blocking communications. The upper bound (*ub*) corresponds to the number of pending communications case when *mcc* statements only complete one of the previous non-blocking communication. Given this approach, a function is incorrect (completion calls missing) if the lower bound of the pnb for the function exit is greater than 0. The function *may* be incorrect if the upper bound of this *pnb* is greater than 0.

**Algorithm 1** Evaluating the number of pending non-blocking communications

1: **function** PNB_BOUND($G = (V, E)$) ▷ $G$: CFG
2: Remove loop backedges in $G$
3: **for** $n \in V$, in topological order **do**
4: **if** $n$ has no predecessor **then** $pnb_n \leftarrow [0, 0]$ ▷ $pnb$: interval of values $[pnb.lb, pnb.ub]$
5: **else**
6: $pnb_n \leftarrow \bigcup_{(p,n)\in E} pnb_p$
7: **end if**
8: **if** non blocking operation $\in n$ **then**
9: $pnb_n \leftarrow [pnb_n.lb + 1, pnb_n.ub + 1]$
10: **end if**
11: **if** unitary completion call $\in n$ **then**
12: $pnb_n \leftarrow [\max(pnb_n.lb - 1, 0), \max(pnb_n.ub - 1, 0)]$
13: **end if**
14: **if** multi completion call $\in n$ **then**
15: $pnb_n \leftarrow [0, \max(pnb_n.ub - 1, 0)]$
16: **end if**
17: **end for**
18: Output *pnb*
19: **end function**

**Extending PARCOACH with Non-Blocking Semantics:**

In Section 1, non-blocking collectives have been shown to share the same constraint as blocking collectives when used together. Once this behavior is identified, supporting non-blocking collectives in the original analysis of PARCOACH is straightforward: the syntactic parser of collective names has to be enhanced with non-blocking ones. Since PARCOACH uses the output list of this step to instrument the problematic nodes, our enhancement will directly benefit from the instrumentation. PARCOACH will then be able to insert functions for runtime checking before all flavor of collectives, including non-blocking ones.

# 3. RESULTS

We tested our two static analyses on the Intel MPI Benchmarks version 4.0 (IMB) and a microbenchmarks suite of unitary tests we created (NBC-bench). These results were computed and averaged with MPICH 3.1.4.

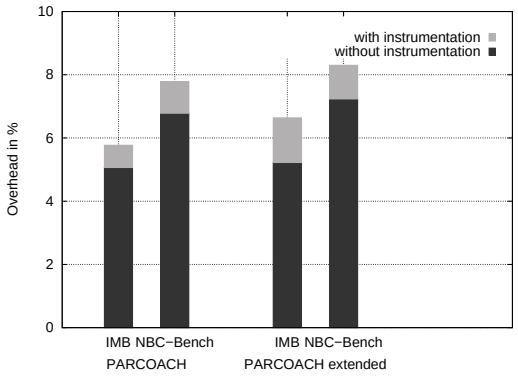


**Figure 1: Overhead of average compilation time with and without verification code generation**

Figure 1 highlights the compilation-time overhead obtained with and without static instrumentation (validation functions insertions) on the two sets of benchmarks. For each test, the overhead of the original PARCOACH analysis, for blocking MPI collective calls, and our improved analysis, checking both blocking and non-blocking MPI collective calls, are provided. The black bars display the original overheads, and the grey bars show the additional overhead created by the addition of non-blocking collectives in the static analysis. Our addition to the checking of collective only bring an extra 1% to the overhead, which remains under 9% of the total compilation time.

# 4. CONCLUSION

In this paper, we extended the static analysis of PARCOACH, originally designed for checking blocking collective calls, to include non-blocking collectives. A second static analysis was included to also check the completion of all non-blocking communications issued in an MPI program. The designed algorithms allow to find if non-blocking calls are not completed in a function, and return the list of problematic nodes for issuing warnings to the developer. This addition brought a small extra overhead compared to the original analysis, and the complete compilation time with the new analyses being less than 10% higher than initial compilation time.

# 5. REFERENCES

[1] D. Dureau and G. Poëtte. Hybrid parallel programming models for amr neutron monte-carlo transport. In *SNA+ MC 2013-Joint International Conference on Supercomputing in Nuclear Applications+ Monte Carlo*, page 04202. EDP Sciences, 2014.

[2] T. Hoefler, P. Gottschling, A. Lumsdaine, and W. Rehm. Optimizing a conjugate gradient solver with non-blocking collective operations. *Elsevier Journal of Parallel Computing (PARCO)*, 33(9):624 – 633, 2007. Selected Papers from EuroPVM/MPI 2006.

[3] 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)*, 28(4):425–434, 2014.