# Transparent Overlapping of Blocking Communication in MPI Applications

Alexis Lescouet, Élisabeth Brunet, François Trahay, Gaël Thomas
*Télécom SudParis,*
*Institut Polytechnique de Paris*
*firstname.lastname*@telecom-sudparis.eu

***Abstract*—With the growing number of cores and fast network like Infiniband, one of the keys to performance improvement in MPI applications is the ability to overlap CPU-bound computation with network communications. While this can be done manually, this is often a complex and error prone procedure. We propose an approach that allows MPI blocking communication to act as nonblocking communication until data are needed, increasing the potential for communication and computation overlapping.**

**Our approach, COMMMAMA, uses a separate communication thread to which communications are offloaded and a memory protection mechanism to track memory accesses in communication buffers. This guarantees both progress for these communications and the largest window during which communication and computation can be processed in parallel.**

**This approach also significantly reduces the hassle for programmers to design MPI applications as it reduces the need to forecast when nonblocking communication should be waited.**

***Index Terms*—MPI; communication; design tools**

## I. Introduction

Nowadays, MPI remains one of the standard runtimes for distributed computing. While the exact proportion varies between applications, in a lot of these, completion time is shared between computation (CPU intensive) and communications between nodes (network intensive). These two components can mostly operate independently, but designing application using the MPI runtime to take advantage of the potential parallelism is hard.

Though nonblocking communications exist in the MPI runtime, these primitives need a corresponding waiting primitive before the communication buffer can be used in order for the application to be semantically correct. The position of these waiting primitives in the application code is the limit before which overlapping computation and communication can happen and is currently at the programmer initiative.

This presents different challenges. Firstly, it is difficult to determine when exactly the waiting primitive should be called in order to make the most of the nonblocking communication. Secondly, as these primitives are statically placed in the application code, it is sometime impossible to take into account performance variations that could have lead to a possible extension of the overlapping period.

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations

To answer these challenges, we propose a new approach to nonblocking communication in which the nonblocking period is dynamically determined by the runtime. This both alleviates the complexity for the developer of the application and allows an optimal overlapping of the communication with the computation.

In our model, rather than using MPI nonblocking primitives, the developer uses seemingly classic blocking communications which are replaced by our versions. When calling a modified blocking primitive, the communication buffer memory is protected to avoid unwanted accesses and the communication is offloaded to a progress thread. The application is then free to do any computation needed while the progress thread takes care of the actual communication until an access occurs to the communication buffer. When this happens the runtime blocks until the buffer is available for use, being then semantically equivalent to a waiting primitive being called.

In this paper, we detail our proposal to new efficient nonblocking communication and show it results in better overlapping of communication and computation while lessening the complexity of application development. More specifically, our method:

- overlaps communication and computation providing up to 73% speedup,
- transparently replaces primitives with efficient ones.

The remainder of the paper is organized as follows. In Section 2 we describe how nonblocking communication are handled currently. In Section 3 we explain the technical details of our approach, COMMMAMA. In Section 4 we evaluate our work with microbenchmarks. In Section 5 we examine related work. In Section 6 we conclude and discuss future work.

## II. Current MPI primitives behavior

In this section, we describe how current MPI options available to developers are either subpar in term of efficiency or complex to use and present available approaches and challenges pertaining to communication and computation overlapping.

### A. Necessary background on MPI semantics

The MPI specification [1] defines an important number of primitives for different use cases. Even for point-to-point communication there is multiple variant of the "send" function: the standard `MPI_Send`, `MPI_Bsend`, `MPI_Ssend` and `MPI_Rsend` and their nonblocking counterparts. The distinction between these different functions depends on two concepts, the message buffering or lack thereof and the synchronous or asynchronous component of the primitive.

Message buffering is the act of copying the data contained in the communication buffer to an alternate buffer in which the data will stay until the communication is complete, allowing the communication buffer to be reused by the application while communication proceeds.

A synchronous communication primitive is one that waits for the data to complete before returning, which means the data is effectively received by the matching receive primitive.

The characteristics of the multiple send primitives are summed up in Table I.

TABLE I
SEND MODES

| | Asynchronous | Buffered |
|---|---|---|
| MPI_Send | Yes | For small message |
| MPI_Bsend | Yes | Yes |
| MPI_Ssend | No | Irrelevant |
| MPI_Rsend | Yes | For small message |

While message buffering allows for a certain degree of communication and computation overlap it also produces an overhead due to copying the communication data to an internal buffer, which becomes costlier when message size increases. Hence the compromise used in multiple MPI implementations to buffer `MPI_Send` data only for small message sizes. In any case, MPI semantics specify that when `MPI_Send` returns, either because the message has been buffered or not, the communication buffer is free to be modified by the application. The buffering mode of `MPI_Send` is commonly known as *eager* while the other mode which needs to wait a matching receive to proceed is known as *rendez-vous*.

Because eager mode already encompasses some overlapping capabilities, our approach only targets rendez-vous mode of `MPI_Send` as `MPI_Rsend` must wait for a matching receive before proceeding and the other primitives are always buffered.

### *B. Blocking and nonblocking are only part of the answer*

Currently, overlapping communication and computation is mostly done using nonblocking communications. The code presented in Listing 1 demonstrates a basic usage of nonblocking communication that would allow overlapping to be done.

However, communication can only run in parallel with computation if there is some kind of progress engine in the runtime implementation. While some approaches to communication and computation overlapping such as [2] propose methods to ensure progress, to the best of our knowledge, there is no production ready progress engine included in well-known MPI implementations for general use cases.

```
void write_function(int* buffer, ...) {
  int* result = do_long_computation();
  memcpy(buffer, result, ...);
}
void example(int* buffer) {
  MPI_Request handle;
  MPI_Isend(buffer, ..., &handle);
  read_only_function(buffer, ...);
  MPI_Wait(&handle, ...);
  write_function(buffer, ...);
}
```

Listing 1. Overlapping using nonblocking communication

Moreover, in Listing 1 `MPI_Wait` is called before the `write_function` because this function does write in the buffer but only after a long independent or read only computation which could also have been overlapped with communication. In general, determining the optimal place to put the waiting primitive can be quite complicated since it can be buried in the call sequence or different depending on conditionals. Considering a progress engine is available, using nonblocking primitives is still complicated and suboptimal.

While Listing 2 is semantically equivalent and much simpler, it presents by itself no opportunity for overlapping.

```
void write_function(buffer, ...) {
  int* result = do_long_computation();
  memcpy(buffer, result, ...);
}
void example(int* buffer) {
  MPI_Send(buffer, ...);
  read_only_function(buffer, ...);
  write_function(buffer, ...);
}
```

Listing 2. Blocking communication equivalent

Currently, blocking and nonblocking both have their drawbacks. By transforming blocking MPI operations into their nonblocking version automatically and managing the overlap period dynamically, our proposal provides at least the efficiency of the nonblocking version while being as simple as the blocking one.

## III. OUR OVERLAPPING INFRASTRUCTURE: COMMMAMA

In this section, we explain how standard calls performed by the application are transformed and treated as we detail the different parts composing COMMMAMA, our system. As presented in Figure 1, we consider a software stack composed of three major elements, the application, COMMMAMA and the MPI runtime.

COMMMAMA is composed of three different layers. The interception layer which intercepts and transforms standard blocking primitives into multiple calls to other layers. The protection layer which manages memory protection for communication buffers and forces the application to wait when an otherwise semantically incorrect memory access occurs. The progress layer which is responsible for communications and offloads communications to a standalone thread. These different components are explained in detail in the next paragraphs.

COMMMAMA operates in the following fashion. When the application invokes a communication primitive, COMMMAMA

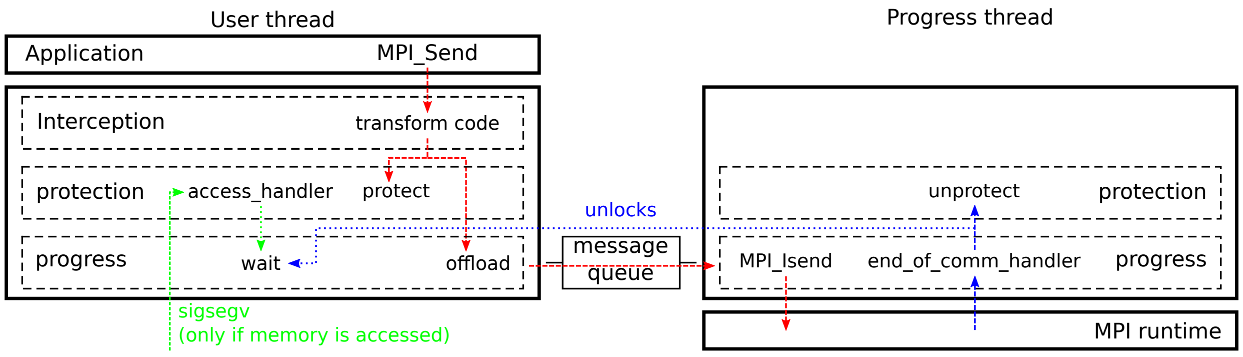

Fig. 1. Overlapping system architecture

intercepts the call, replaces it with its nonblocking equivalent and protects the communication buffer to catch forbidden memory access patterns. For *MPI_Send* the buffer becomes read only while for *MPI_Recv*, the buffer becomes write only. COMMMAMA then returns to the normal control flow of the application and makes communications progress in the background. When the application accesses a communication buffer with an unwanted access type, either a write for *MPI_Send* or a read for *MPI_Recv*, COMMMAMA waits until the communication has ended to return control to the application.

## A. The interception layer

We chose to avoid modifying directly any MPI runtime to be compatible with any implementation and to avoid maintaining a forked version. Therefore, COMMMAMA uses a shared library which replaces needed MPI functions by custom versions.

Our interception library first stores all the existing MPI library functions into dedicated function pointers and then declares functions with their original names. This library is then loaded before the real MPI library using *LD_PRELOAD*, effectively renaming MPI library symbols and replacing them with ours. To ease the use of this shared library, we provide an executable managing this configuration and using the interception library becomes as simple as `mpi_interceptor mpirun [args]`.

On top of being easy to use, this method has another upside which is the possibility to use it on already compiled or written software, without any need to modify the code to obtain its full efficiency.

## B. The offloading subsystem

The offloading subsystem provides asynchronous progress for MPI communications. It consists of a standalone thread which is spawned when the MPI runtime is initialized and a queue in which are registered requests to be treated by the offloading thread.

The offloading thread polls the queue for any new request to perform, if none are currently waiting, the thread performs calls to the MPI runtime to make pending communication progress. When a request completes according to the MPI runtime, the offloading thread invokes a callback function and passes the request data (called `end_of_comm_handler` in the rest of this paper).

This architecture is beneficial for two reasons. All real MPI runtime function called being nonblocking, the progress can easily be done using a nonblocking test function for all ongoing communications at once, `MPI_Testsome`. More importantly, this allows faster or smaller communications to be finished as soon as possible, independently of their starting order.

This offloading subsystem only takes care of communication progress on its own but the callback executed at the end of communication allows for different behavior to be added.

## C. The protection mechanism

The protection mechanism uses the interface provided by the offloading subsystem to enable a memory protection mechanism on the communication buffer and disable it when the communication buffer can be accessed freely. This corresponds in classic nonblocking communication to the end of the `MPI_Wait` function.

The protection framework provides two primitives respectively to enable and disable protection on a specific buffer. Both of these functions internally call the POSIX `mprotect` function which allows a program to restrict access to its own address space. This restriction is a combination of the following access modes: read, write, execute. When memory protected with `mprotect` is accessed using a forbidden mode, the operating system raises a segmentation fault signal to the application which in turn can be intercepted in order to treat the problem.

Our custom handler for this segmentation fault signal checks whether the memory address on which a bad access occurred is a zone managed by the protection mechanism and if so calls a modifiable callback function (called `access_handler` in the rest of this paper).

## D. Making blocking communication nonblocking

Using the two components described above, we built our overall system by making:

- `end_of_comm_handler` disable memory protection on the buffer when the corresponding communication has ended,

- `access_handler` wait until communication has ended,
- our custom `MPI_Send` enable protection and offload the communication to our system.

This configuration guarantees two important behaviors, every buffer on which memory protection is enabled during the call to `MPI_Send` will see its protection disabled eventually and any unwanted access to the memory will cause the application thread to wait as though a real call to `MPI_Wait` was issued. Therefore, the period during which communication is overlapped with computation is dynamically decided by the first forbidden access to the buffer, which in turns corresponds to the optimal limit we can afford according to the MPI specification.

Moreover, while this is not the goal of COMMMAMA, the offloading subsystem uses nonblocking calls internally and can thus be used alone to provide asynchronous progress to nonblocking primitives in addition to providing progress to blocking calls using the protection mechanism.

## IV. EVALUATION

In the section we evaluate the performance of our approach using a number of benchmarks.

### A. Evaluation test bed characteristics

All the experiments are done using the Grid'5000 [3] *grimoire* cluster. Nodes of this cluster consist of dual socket 8 core Intel® Xeon® E5-2630 v3 with 128 GB memory. These nodes are connected using 56 Gb/s Mellanox Technologies ConnectX-3 InfiniBand network adapters. We use GCC 8.3.0 and OpenMPI 3.1.3 [4].

All the results presented thereafter are produced using this test bed by running the mentioned benchmark 2000 times and taking a mean value.

### B. Base overhead analysis

In this section we compare our approach, COMMMAMA to a baseline version using only *openmpi* to study the effect of our system on base performances. This is done using a ping-pong microbenchmark with no computation to exhibit the overhead introduced by our method.

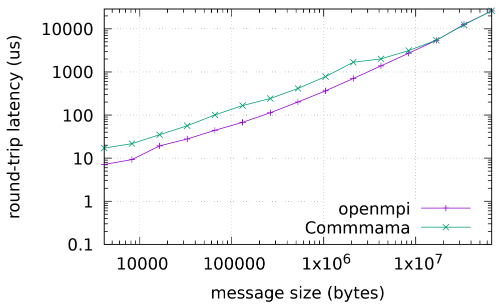

Fig. 2. Round-trip latency with no compute

Figure 2 represents the round-trip latency of messages of varying sizes, both axes are represented using log scale. Depicted on this figure, the overhead incurred by COMMMAMA is linear until important message sizes where it becomes negligible. This is mostly due to the cost of the protection subsystem and more specifically the linear cost of the `mprotect` system call.

This overhead is indeed due to the modification of the page table structure during the `mprotect` call used to enforce memory protection on the process address space. With the growth of message size, the range of memory protected by the system call grows which in turns increases in a linear fashion the number of page structure to modify. However, this overhead is not as important as to impact the capacity to overlap communication and computation as demonstrated in the next subsection.

### C. Overlap capabilities evaluation

To evaluate the communication and computation overlap provided by our design, we use a modified version of the ping-pong microbenchmark with added computation time between communication phases.

This benchmark executes as follows. Process rank 0 calls `MPI_Send`, computes for a given time, writes one byte to the buffer then calls `MPI_Recv`, computes for a given time and reads one byte from the buffer. Process rank 1 does the same but starting by `MPI_Recv`, followed by the compute period and the read operation then `MPI_Send`, the compute period and the write operation.

In this second benchmark the memory operation plays an important role as it ensures the previous communication (either send or receive) is terminated before continuing. This simulates the first access to buffer data in the computation period following a communication.

The expected execution timelines are presented in Figure 3.

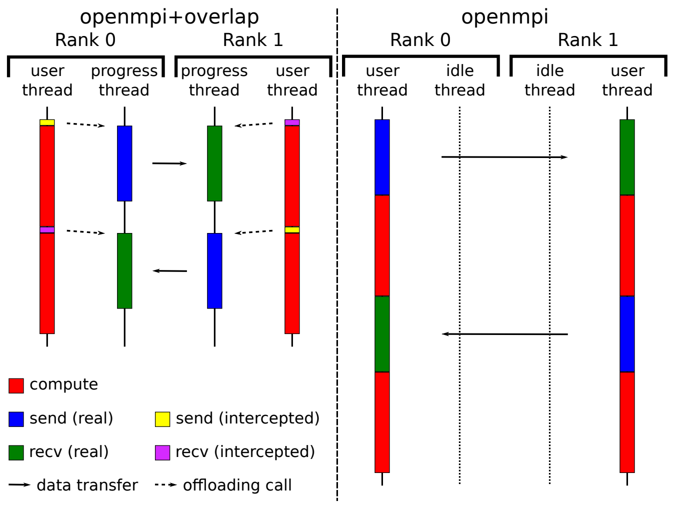

Fig. 3. Expected execution timeline for microbenchmark

The results presented thereafter use this benchmark with a varying amount of computation time, respectively $2 \times 200$, $2 \times 2000$ and $2 \times 20000$ microseconds. The *openmpi* curve represents the default OpenMPI behavior, the COMMMAMA curve represents the behavior of our approach on top of the same OpenMPI runtime and the *optimal completion time* curve represents the theoretical optimum which is $\max(t_{compute}, t_{comm})$, where $t_{compute}$ is the total time of computation (i.e. the sum of both occurrences in the microbenchmark) and $t_{comm}$ is the time of completion of the communication for the default OpenMPI implementation taken from Section IV-B.

Round-trip latencies in Figure 4 all demonstrate two different behaviors. Between 4KB and a certain value called $s_{threshold}$ below, COMMMAMA remains very close of the optimal completion time, while *openmpi* is equal to $t_{compute} + t_{comm}$. Finally, between $s_{threshold}$ and 64MB, both *openmpi* and COMMMAMA increase gradually, with COMMMAMA staying close to the optimal value.

The default OpenMPI implementation cannot overlap any communication with computation thus resulting in an increased completion time. This is particularly visible when the communication time and computation time are close. On the contrary, COMMMAMA performs well, showing a high overlapping capability. This second behavior persists until message size attains $s_{threshold}$, the message size until which $t_{comm} < t_{compute}$.

As for the second phase, when message size exceeds $s_{threshold}$, both *openmpi* and COMMMAMA round-trip latency start increasing, with COMMMAMA still being close to the optimal completion time value which is now $t_{comm}$ as exceeding $s_{threshold}$, $t_{comm}$ has become the longest task required for completion.

Favoring communication and computation overlap, our approach can really shine when their is enough computation to overlap communication. As shown by speedups plots in Figure 4 speedup can be as high as 73% and is good in any case when $t_{compute}$ is slightly bigger than $t_{Comm}$. For small communications, with a small compute time of 400 microseconds, speedup attains 42%, with 4 milliseconds of compute time, speedup attains its best value for medium-sized communications, 73% for 1MB messages. After this point the speedup decreases due to the compute time becoming unsufficient to correctly overlap communication for bigger messages. With 40 milliseconds of compute time, big communications are at their best with up to 50% speedup.

## V. RELATED WORK

The field of asynchronous progress for MPI primitives has been extensively studied.

Some approaches use compile time techniques to evaluate the best position for nonblocking primitives and specifically `MPI_Wait` calls.

This is the case of [5] which uses source to source translation to remodel code using compiler pragmas to divide the code into three types of regions: send, receive and compute. While this allows some communication to be overlapped with computation by running communication and compute region in parallel on different cores, the compute region still need to start with a waiting primitive (in most cases `MPI_Waitall`) which in turns seems too coarse grained to attain a satisfying overlap ratio in complex cases.

L. Fishgold et al. [6] propose a finer grained, automated technique which detects patterns presenting an opportunity for tranformation by parsing the source code and then modify it. This approach focuses on finding compute loops filling parts of communication buffers at each iteration and modifying them to send these parts when ready. While this technique covers a large portion of MPI applications, especially High Performance Computing related ones, our approach offers a broader field of application. Moreover, they focus on `MPI_Alltoall` collective while our work does not target a specific communication primitive.

A similar approach from Jichi Guo et al. [7] uses static code analysis to replace blocking communications by nonblocking communications in loop based computations thus overlapping communication step $i$ with computation step $i + 1$. While this approach is close to our proposal, it still uses compile time analysis, compute loops and does the communication progression using `MPI_Test` primitives inserted in the loop body. On the contrary, our progress thread allows for a better degree of parallelism to be attained and works on broader patterns than compute loops. This approach is closely related to [8].

Another attempt to increase the overlap of communication and computation is presented by Anthony Danalis et al. [9]. They describe an algorithm used to optimize the overlap window at compile time by replacing blocking primitives with nonblocking ones and placing the waiting primitive as far as possible to increase performances. However, unlike previous work they use the assumption that the compiler is aware of MPI semantics. While this allows for an even better increase in the overlap window, this makes the work of compiler developers and MPI developers more complicated, which is not the case in our approach as it operates transparently. This is similar to [10] which uses clang to perform modifications.

The method presented in [11] uses code instrumentation and offline analysis to determine code fragments where overlapping capabilities are not fully exploited. After the instrumented run has terminated, their tool uses the trace files to perform optimizations.

Other approaches use runtime techniques. This is the case of [12] and [13] that use threads to increase asynchronous progress by running communications in separate tasks.

Karthikeyan Vaidyanathan et al. [2], probably the closest approach to ours, delegates MPI nonblocking calls to a standalone thread. Our approach builds upon and extends this approach by using the memory protection mechanism to transform blocking MPI calls into nonblocking equivalents. While we agree on the potential gain provided by such a method, such a gain can only be obtained if waiting primitives are placed correctly. Our approach solves this problem by dynamically finding the optimal (i.e. latest) moment to wait.

Similarly, Min Si et al. [14] present an asynchronous progress system using processes in place of threads and is targeted at many-core architectures.

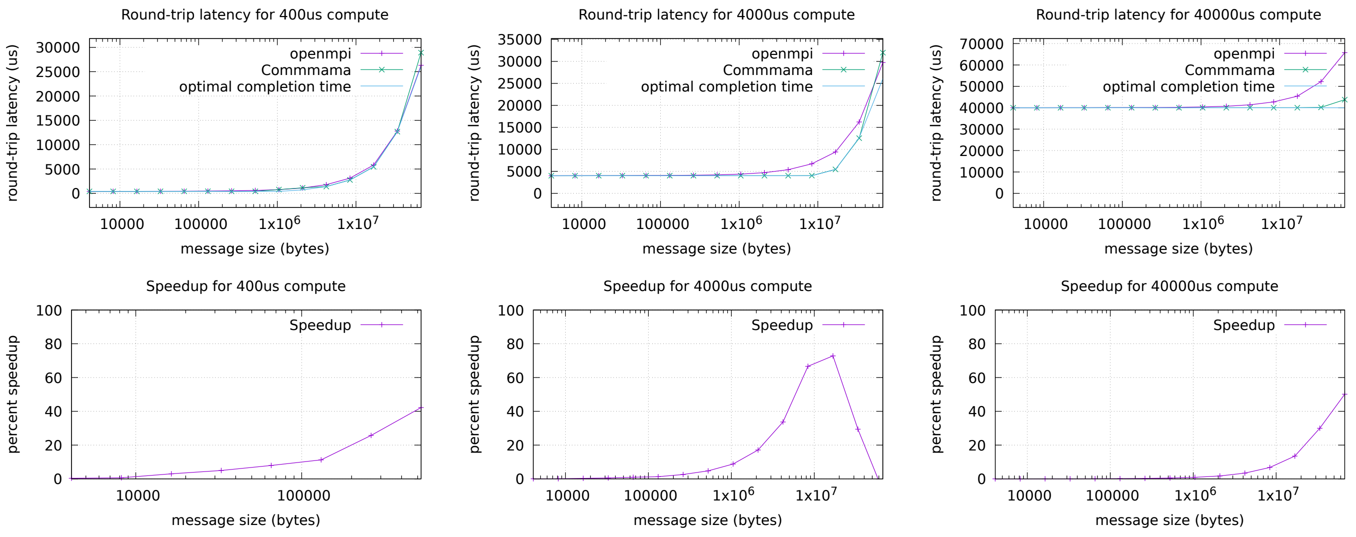

Fig. 4. Round-trip latencies and speedups for $t_{compute} = 400$, 4000, 40000us

Sylvain Didelot et al. [15] propose a different approach, with no added parallel execution unit, which uses idle cycles created by imbalance in computation time to improve progression.

## VI. Conclusion and future work

In this paper, we describe a new method to increase communication and computation overlapping in MPI applications. Our approach combines an interception mechanism, a standalone progress thread and a protection mechanism. This allows blocking primitives to be transparently replaced by our mechanism which transfers the communication responsibility to our standalone thread while protecting the application buffer memory to intercept future read or write. As progress is ensured by the thread, the memory protection is used to monitor memory accesses and let the application benefit from automated communication and computation overlap for an optimal amount of time. Depending on the ratio of computation and communication in the application, we show that our approach can provide up to 73% speedup.

While the amount of overlap time gained from our method is optimal if considering the buffer in its entirety, future work may include splitting a given blocking communication into a number of nonblocking communications, thus allowing computation to continue for other buffers when only some are accessed. On another note, extending our approach to MPI blocking collectives is already part of future work as it should also enhance communication and computation overlapping periods in numerous cases.

## References

[1] "Mpi-3.1 standard," https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf, 2015.

[2] K. Vaidyanathan, D. D. Kalamkar, K. Pamnany, J. R. Hammond, P. Balaji, D. Das, J. Park, and B. Joó, "Improving concurrency and asynchrony in multithreaded MPI applications using software offloading," in *SC*. ACM, 2015, pp. 30:1–30:12.

[3] "Grid'5000 website," https://www.grid5000.fr.

[4] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104.

[5] S. M. Martin, M. J. Berger, and S. B. Baden, "Toucan - A translator for communication tolerant MPI applications," in *IPDPS*. IEEE Computer Society, 2017, pp. 998–1007.

[6] L. Fishgold, A. Danalis, L. L. Pollock, and D. M. Swany, "An automated approach to improve communication-computation overlap in clusters," in *PARCO*, ser. John von Neumann Institute for Computing Series, vol. 33. Central Institute for Applied Mathematics, Jülich, Germany, 2005, pp. 481–488.

[7] J. Guo, Q. Yi, J. Meng, J. Zhang, and P. Balaji, "Compiler-assisted overlapping of communication and computation in MPI applications," in *CLUSTER*. IEEE Computer Society, 2016, pp. 60–69.

[8] K. Murthy and J. Mellor-Crummey, "A compiler transformation to overlap communication with dependent computation," in *PGAS*, 2015.

[9] A. Danalis, L. L. Pollock, D. M. Swany, and J. Cavazos, "Mpi-aware compiler optimizations for improving communication-computation overlap," in *ICS*. ACM, 2009, pp. 316–325.

[10] V. M. Nguyen, E. Saillard, J. Jaeger, D. Barthou, and P. Carribault, "Automatic code motion to extend MPI nonblocking overlap window," in *ISC Workshops*, ser. Lecture Notes in Computer Science, vol. 12321. Springer, 2020, pp. 43–54.

[11] E. Saillard, K. Sen, W. Lavrijsen, and C. Iancu, "Maximizing communication overlap with dynamic program analysis," in *HPC Asia*. ACM, 2018, pp. 1–11.

[12] A. Denis, "pioman: A pthread-based multithreaded communication engine," in *PDP*. IEEE Computer Society, 2015, pp. 155–162.

[13] F. Trahay, A. Denis, O. Aumage, and R. Namyst, "Improving reactivity and communication overlap in MPI using a generic I/O manager," in *PVM/MPI*, ser. Lecture Notes in Computer Science, vol. 4757. Springer, 2007, pp. 170–177.

[14] M. Si, A. J. Peña, J. R. Hammond, P. Balaji, M. Takagi, and Y. Ishikawa, "Casper: An asynchronous progress model for MPI RMA on many-core architectures," in *IPDPS*. IEEE Computer Society, 2015, pp. 665–676.

[15] S. Didelot, P. Carribault, M. Pérache, and W. Jalby, "Improving MPI communication overlap with collaborative polling," in *EuroMPI*, ser. Lecture Notes in Computer Science, vol. 7490. Springer, 2012, pp. 37–46.