A methodology for assessing computation/communication overlap of MPI nonblocking collectives

Alexandre Denis, Julien Jaeger, Emmanuel Jeannot, Florian Reynier

▶ **To cite this version:**

Alexandre Denis, Julien Jaeger, Emmanuel Jeannot, Florian Reynier. A methodology for assessing computation/communication overlap of MPI nonblocking collectives. Concurrency and Computation: Practice and Experience, 2022, 34 (22), 10.1002/cpe.7168 . hal-03922777v1

HAL Id: hal-03922777

https://inria.hal.science/hal-03922777v1

Submitted on 4 Jan 2023 (v1), last revised 4 Jan 2023 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

RESEARCH ARTICLE

A Methodology for Assessing Computation/Communication Overlap of MPI Nonblocking Collectives

Alexandre Denis*[1] | Julien Jaeger[2,3] | Emmanuel Jeannot[1] | Florian Reynier[1,2]

[1]Inria, LaBRI, Univ. Bordeaux
[2]CEA,DAM,DIF, F-91297 Arpajon, France
[3]Laboratoire en Informatique Haute Performance pour le Calcul et la simulation, Université Paris-Saclay, CEA, France

Correspondence
*Corresponding author,
Alexandre DENIS
Inria
200 avenue de la vieille tour
F-33405 Talence Cedex, FRANCE
Email: Alexandre.Denis@inria.fr

By allowing computation/communication overlap, MPI nonblocking collectives (NBC) are supposed to improve application scalability and performance. However, it is known that to actually get overlap, the MPI library has to implement progression mechanisms in software or rely on the network hardware. These mechanisms may be present or not, adequate or perfectible, they may have an impact on communication performance or may interfere with computation by stealing CPU cycles. From a user point of view, assessing and understanding the behavior of an MPI library concerning computation/communication overlap is difficult.
In this paper, we propose a methodology to assess the computation/communication overlap of NBC. We propose new metrics to measure how much communication and computation do overlap, and to evaluate how they interfere with each other. We integrate these metrics into a complete methodology. We compare our methodology with state of the art metrics and benchmarks, and show that ours provides more meaningful informations. We perform experiments on a large panel of MPI implementations and network hardware and show when and why overlap is efficient, nonexistent or even degrades performance.

KEYWORDS:
High-performance computing, MPI, collective communication

1 | INTRODUCTION

In order to execute parallel applications on large-scale supercomputers efficiently, applications are often programmed using the Message Passing Interface (MPI) standard[1]. MPI uses the SPMD (Single Program Multiple Data) model to describe how each process composing the applications are communicating. MPI features diverse primitives such as point-to-point communications or collective operations. Collective operations are a set of abstract routines enabling a group of processes to execute in an efficient and concise way communications on a group with an arbitrary number of processes. Since the version 3.0 of the MPI standard, all the collective operations of MPI have a nonblocking version.[1] ~~version.~~

Using nonblocking collectives allows the application to execute independent computations between the collective initiation call and the completion call (*e.g.*, `MPI_Wait`). Thanks to this, it should be possible to hide the communication behind computation and to save time through computation/communication overlap. However, for such overlap to effectively happen, it is required that the communication *progresses* in the background while the application continues to execute its computation. Without this background progress, the MPI library is not able to execute the chosen collective algorithm and the data transfer can only happen when the application explicitly calls other MPI routines.

[1]In this paper, we use the spelling *nonblocking*, without hyphen, as in the MPI specification[1].

Implementing a progression engine for collective operations is much more difficult than for point-to-point communication as such collective may require to implement a complex algorithm[2] (*e.g.*, diffusion tree) involving computation (*e.g.*, for reducing values). Hence, MPI library developers are struggling to implement such efficient progress engines. Therefore, in the literature, up to now, few applications have been reported to use MPI nonblocking collective[3].

The goal of this paper is to provide a methodology to assess the quality of the progression engines of different MPI implementations, and to understand the cause of bad performance when such case occurs. To perform this experimental study, we first detail a set of metrics and procedures to measure computation/communication overlap in the case of collective routines. The goal of these metrics is not only to measure if such overlap actually takes place but also what is the impact of the progress engine on the computation as well as the interaction between the MPI runtime system and the application. Based on this set of metrics, we evaluate our methodology on different nonblocking collective routines and on several popular MPI implementations to assess how the communication/computation overlap behaves. We also compare our metrics with existing ones, to outline the limitation of the state of the art.

This paper is organized as follows. In Section 2 we describe the related work. The different metrics for measuring overlap and its impact on performance are presented in Section 3. Our methodology to measure the performance of a nonblocking collective communication is detailed in Section 4. The implementation of the benchmark used for the experiments is described in Section 5. Section 6 details two representative cases of our methodology and compares it with other benchmarks. Section 7 offers a survey of results on a variety of MPI implementations and networks. Last, we conclude and give our final remarks in Section 8.

2 | RELATED WORKS

Nonblocking communications have been available in MPI for point-to-point communication since the first version of the standard[4] for more than two decades. While the progress of point-to-point communication is not straightforward[5], the advent of nonblocking collectives[1] has pushed asynchronous progression to its limit. It has been demonstrated that progress threads are a valid solution for a good overlap[6,7,8]. Since a progress thread will generate communications[9], work on progression meets work on multi-threaded MPI[10,11,12,13,14].

In this paper, we want to assess how these principles are effectively utilized in current MPI implementations to improve progression of MPI nonblocking communications. Several benchmarks already exist. The Intel MPI Benchmarks (IMB)[15] use a combination of an overlap ratio and the pure communication duration to evaluate MPI nonblocking collective performance. OSU microbenchmarks[16] propose a similar approach with the addition of possible GPU computation. NBCBench[9] uses a similar overlap measurement, but it focuses on the measurement of one issued collective at a time, instead of pipe-lining multiple collectives. It also divides the communication time in two parts: an overlappable time and a non-overlappable time, which allows measuring the overlap efficiency only on the overlappable time. The benchmark was later updated to also check the CPU overhead[17], based on lessons learned from the SkaMPI benchmarks[18] which does not handle nonblocking collectives. However, these overlap metrics are not precise enough. For example, the OSU benchmarks give an overlap ratio based on the mean of all ranks involved in the collective. This can be flawed as tree-based algorithms may induce very different workloads on each rank, and the mean will not be representative. Moreover, the use of `MPI_Barrier` to synchronize the MPI processes does not imply that each rank will start the measured collective at the same time, as the MPI processes may not leave the `MPI_Barrier` simultaneously. MadMPI benchmark[5] relies on an *overhead ratio* to measure the overlap of point-to-point communications. This overhead ratio is a more flexible metric than the overlap ratio used in other benchmarks, as it is usable even when communication and computation do not take the same time, and it highlights cases where the overlap slows down computations or communications.

Clock synchronization[19] is a common issue that is encountered when designing an MPI benchmark. It has been demonstrated[20,21] that correcting clock offsets is not sufficient to keep a good synchronization: the time of the CPU clocks may diverge between nodes, and skewness needs to be taken into account. Timing issues can also come from the benchmark configuration. If multiple MPI processes are located on the same core through virtualisation, using traditional MPI timer calls (e,g,. `MPI_Wtime`) will not measure the time on the caller MPI process as expected, but may measure the cumulative time of the co-located MPI processes[22]. Hence, measurement method should be carefully chosen, and compatible with the benchmarks design.

While several benchmarks exist for evaluating nonblocking collective communication, none of them gather all the necessary data to interpret bad overlap. In our benchmark, we propose our own metrics so as to be able to evaluate nonblocking communications and diagnose cases of bad overlap.

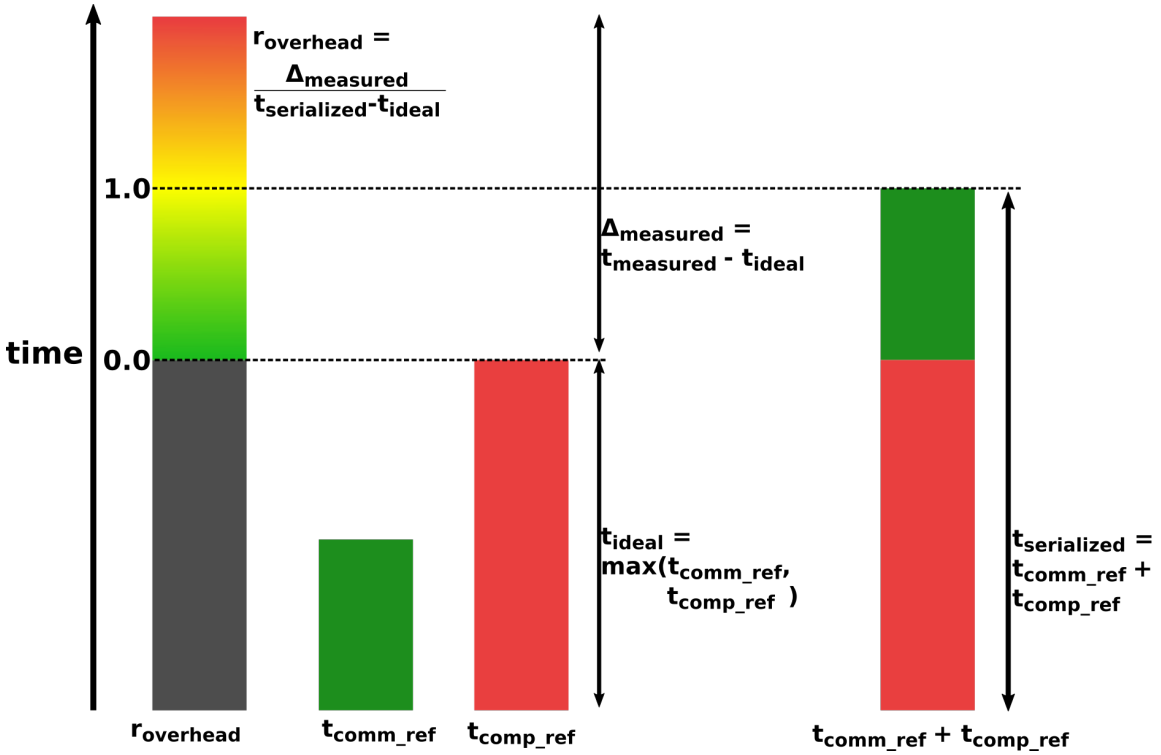

FIGURE 1 Definition of the overhead ratio

3 | METRICS FOR OVERLAP AND ITS IMPACT ON PERFORMANCE

In this section, we define our metrics to assess communication / computation overlap, and to measure the impact of overlap on both communication and computation performance.

3.1 | Measuring Overlap

To measure overlap, we use the *overhead ratio* metric already defined[5] for point-to-point communication. This metric shows the overhead of the actual performance when overlapping, compared to the ideal case with perfect overlap. This ratio is actually not specific to point-to-point communication and can be extended for nonblocking collectives too. We will see in Section 4.1 how we measure time for operations that involve more than two nodes.

The *overhead ratio* is defined as follows and as depicted in Figure 1. Consider $t_{\text{comp\_ref}}$ the reference time of computation and $t_{\text{comm\_ref}}$ the reference time of communication, without overlap. Then, in case of overlap, the ideal total time assuming perfect overlap is:

$$t_{\text{ideal}} = \max(t_{\text{comp\_ref}}, t_{\text{comm\_ref}}).$$

The *overhead* of the actual measured performance is then:

$$\begin{aligned}\Delta_{\text{measured}} &= t_{\text{measured}} - t_{\text{ideal}} \\ &= t_{\text{measured}} - \max(t_{\text{comp\_ref}}, t_{\text{comm\_ref}}).\end{aligned}$$

This value is an absolute time, hard to interpret since it depends on network speed and computation time. We *normalize* it relative to the serialized case, *i.e.*, the case where computation and communication are run in sequence without overlap. The

overhead for the serialized time is defined as:

$$\begin{aligned}\Delta_{\text{serialized}} &= t_{\text{serialized}} - t_{\text{ideal}} \\ &= t_{\text{comm\_ref}} + t_{\text{comp\_ref}} - \max(t_{\text{comp\_ref}}, t_{\text{comm\_ref}}) \\ &= \min(t_{\text{comp\_ref}}, t_{\text{comm\_ref}}).\end{aligned}$$

We thus define the *overhead ratio* as:

$$r_{\text{overhead}} = \frac{\Delta_{\text{measured}}}{\Delta_{\text{serialized}}}$$

and thus we get:

$$r_{\text{overhead}} = \frac{t_{\text{measured}} - \max(t_{\text{comp\_ref}}, t_{\text{comm\_ref}})}{\min(t_{\text{comp\_ref}}, t_{\text{comm\_ref}})}. \quad (1)$$

In case no overlap happens and computation and communication have been done sequentially, we have $t_{\text{measured}} = t_{\text{comp\_ref}} + t_{\text{comm\_ref}}$ and thus $r_{\text{overhead}} = 1$. In the case of perfect overlap, we have $t_{\text{measured}} = \max(t_{\text{comp\_ref}}, t_{\text{comm\_ref}})$, and thus $r_{\text{overhead}} = 0$. A ratio greater than 1 means that t_{measured} is slower than sequential execution, which may happen in case of interference between computation and communication. Finally, in some cases, we get a negative ratio, which happens if the measured time is faster than expected for perfect overlap; it is mostly observed when there are imprecisions in the measure of reference time for computation and/or communication.

3.2 | Interference between Computation and Communication

Low performance when overlapping computation and communication is not always a sign of bad communication progression in the background. It may be caused by performance degradation of either computation or communication (or both!) because of contention or interferences. It may be especially the case when the mechanisms used for communication progression steal CPU cycles from computation.

While all cases of bad overlap performance will be captured by the *overhead ratio* defined in the previous section, this ratio cannot show whether the degradation is caused by lack of communication progression or by computation or communication slowdown. Thus, we propose an additional metric that will help diagnose the cause of low overlap performance.

3.2.1 | Impact of Inactive MPI Runtime on Computation

First, the MPI runtime system itself may have an impact on computation even without any communication, by the sole effect of the network polling mechanisms running in the background. It may be especially the case with MPI libraries that rely on an asynchronous progress thread for communication progression. Such a thread will use CPU time, which may slow down computation or cause load imbalance.

To quantify the slowdown of the application caused by background MPI execution, we propose a metric called the *MPI impact ratio*. We define it as the ratio between the computation time with and without the MPI library loaded and initialized.

To measure it, we first measure the reference computation time without MPI $t_{\text{comp\_ref}}$. Then, we run the exact same computation code with the addition of enclosing `MPI_Init` and `MPI_Finalize`, and compiled and linked with the `mpicc` wrapper provided by the tested MPI implementation. We then get $t_{\text{comp\_passive\_mpi}}$, the computation time with the MPI runtime. The time is computed by running the largest square matrix multiplication in a given time on one core. Each core determines its own time by running one OpenMP thread. We define the *MPI impact ratio* as follows:

$$r_{\text{MPI\_impact}} = \frac{t_{\text{comp\_passive\_mpi}}}{t_{\text{comp\_ref}}}. \quad (2)$$

If the presence of the MPI runtime causes no performance penalty, the ratio is 1. Otherwise, a ratio higher than 1 indicates that the active MPI runtime impacts the computation.

3.2.2 | Impact of Active Progression and Computation Concurrency

Actual background progression of communication is likely to have more impact on computation than only polling. Running nonblocking collective operations involve more than data transfer that can be offloaded to the Network Interface Card (NIC): it

executes the collective algorithm. If the NIC does not directly support the offload of collective communications[23], the algorithm will take CPU time to execute, and it must be periodically scheduled to trigger all the transfers at the right time. Thus, it is expected to actually consume more CPU time than progression of point-to-point nonblocking communication.

To assess this phenomenon, we propose a metric to measure the slowdown caused on computation by the progression of communication in the background. Not all applications are expected to suffer from the same impact, we can nonetheless evaluate the runtime tendency to disturb the application, and conversely the communication disturbed by computation.

We suppose we have a fixed amount of computation between a nonblocking MPI call and the related `MPI_Wait`, and measure the time spent in each part of the execution, as depicted in Figure 2:

- t_{call} the time taken by the nonblocking MPI call itself,
- t_{comp} the time of the compute function with communication running in the background,
- t_{wait} the time spent in `MPI_Wait` after computation. It must be noted that if communication did not progress in the background at the same time as computation, this time may be significantly high.

FIGURE 2 Definition of times with overlap

To evaluate the slowdown of computation and communication caused by overlap, we compare these times with the reference time of computation and communication without overlap. We use the same reference values as defined in Section 3.1, namely t_{comm_ref} the collective communication time without overlapping computation, and t_{comp_ref} the computation time without overlapping communication, as depicted in Figure 3.

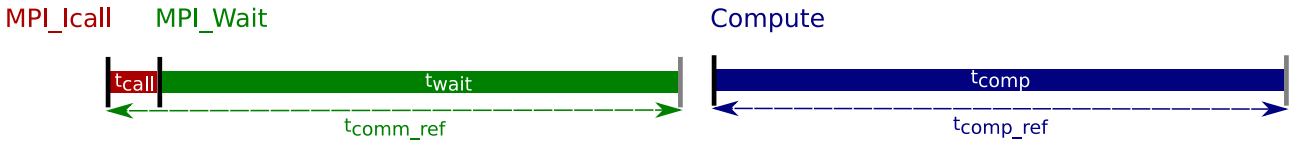

FIGURE 3 Definition of reference times

To measure the *slowdown* of computation in the presence of overlapping communication, we define the *computation slowdown ratio* as:

$$r_{comp\_slowdown} = \frac{t_{comp}}{t_{comp\_ref}}. \tag{3}$$

This ratio measures the time taken by the computation to complete while the communication is running. It does not tell whether the computation is actually slower (because of contention, etc.) or whether CPU cycles have been stolen by the MPI library to make communication progress hence delaying the end of the computation. A slowdown in the computation will not necessarily lead to worst performance than without using nonblocking operations, since it may be counterbalanced by the time gained thanks to an efficient overlap, *i.e.*, less time spent in the `MPI_Wait` function to finish the communication.

We couple this computation slowdown ratio with another metric used to evaluate the time spent in communication primitives compared to the reference situation. We use the exact same principle as for computation slowdown: we compare the time spent in MPI primitives $t_{call} + t_{wait}$ in the overlap case, with the communication time without overlap t_{comm_ref}. Thus, we define r_{comm} as:

$$r_{comm} = \frac{t_{call} + t_{wait}}{t_{comm\_ref}}. \tag{4}$$

It is expected that, in the case of overlap, most of the communication is done in the background at the same time as computation. Thus, the time taken by MPI primitives should be shorter. Since this ratio measures the time spent in the MPI primitives, in case of good overlap, r_{comm} is expected to be significantly lower than 1 thanks to a much lower t_{wait}. A ratio greater than 1 is always the sign that communication is slower than without overlap.

Finally, a ratio around 1 can have two explanations. The first reason to have such ratio is that no communication overlap happened, hence the time for the communication t_{comm_ref} is directly split between the initiation call time t_{call} and the completion call t_{wait}. It must also be noted that a ratio around 1 may reveal that communication is slower than without overlap, caused by interference with computation. In this case, the extra time taken by the communication is hidden in the computation time due to actual overlap, hence it is not included in $t_{call} + t_{wait}$.

3.3 | Using these Metrics to Diagnose Bad Overlap

In this section, we will show how the metrics we just defined may be used to diagnose cases of bad computation/communication overlap. Bad overlap is defined by an overhead ratio, as given by Equation (1), greater than 0. With good overlap, we expect $r_{comp\_slowdown} \approx 1$ and $r_{comm} \approx 0$.

3.3.1 | No pProgression

The most common case of bad overlap happens when the MPI library does not have any progression mechanism to make communication actually progress in the background. The same behavior may happen with an MPI library that features a progression thread for communications, but because of thread placement, the progression thread was not scheduled at the right time and thus did not have the opportunity to make communication progress.

Without any communication progression running alongside the computation function, the whole communication is ultimately done in the `MPI_Wait` call, which results in an execution as depicted in Figure 4. In this case, $r_{comm} \approx 1$ as defined in Equation (4), which is roughly the same situation as in the *serialized* case, despite the use of nonblocking communications. In this scenario, nothing disturbs computation and thus $r_{comp\_slowdown} \approx 1$.

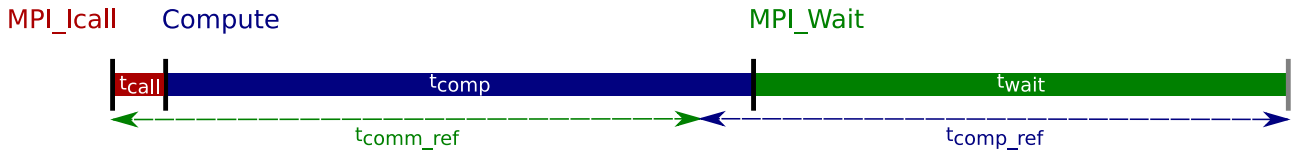

FIGURE 4 Protocol execution with no overlap

In some cases, even without progression mechanisms, some overlap happens thanks to DMA: the first step of the collective algorithm is started in the MPI call and is executed by the NIC in the background. However, the following steps in the algorithm need the CPU to be performed, and they will be started only once the communication is done, hence once the CPU is free, if there is no progression. In this case, r_{comm} is a little bit lower than 1, but converges to 1 for large number of nodes, while overlap becomes negligible.

3.3.2 | Computation sSlowdown

Another cause of inefficient overlap is when the progression mechanism actually does overlap, but hinders the computation to a point where the overall time is higher than serialized computation and communication. The main clue to detect this problem is looking up the impact on the computation overhead: since the progression takes CPU time, it slows down the computation, which results in $r_{comp\_slowdown} > 1$.

Since communication progresses alongside the computation, in this case communication is already over when we reach `MPI_Wait` and thus we have $r_{comm} \approx 0$. This case is depicted in Figure 5.

The bottom line is: in the presence of communication progression mechanisms, computation is slower, and thus overlap may not be worth it.

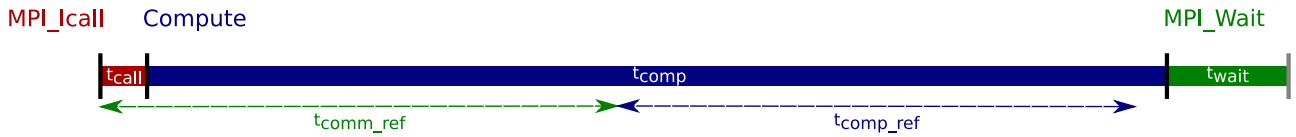

FIGURE 5 Protocol execution with communication significantly hindering computation

3.3.3 | Contention without oOverlap

The last possible case is both $r_{\text{comp\_slowdown}} > 1$ and $r_{\text{comm}} > 1$. A possible scenario for this case is communication was serialized, without any overlap, but at the same time computation was slowed down. MPI process imbalance with an aggressive progress thread can lead to such a result. If the progression thread is scheduled very often, it will greatly delay the completion of the computation. However, it is possible that while being scheduled, the progress thread has nothing to progress. If the mandatory first operation of the algorithm on the local rank is to receive data, the progress thread may wait for these data each time it is scheduled. In case of MPI process imbalance, the data might be sent very late by the other MPI rank. If the expected data arrive once the computation is done on the local MPI process, then the whole collective communication will happen without any overlap, and the computation time would have been hindered by the progress thread.

4 | OUR METHODOLOGY FOR ASSESSING COMPUTATION/COMMUNICATION OVERLAP OF NBC

In this section, we discuss some technical aspects of our methodology to measure performance and overlap on nonblocking collectives.

4.1 | Multiple MPI Processes Time Variation

By design, collective operations usually involve more than two MPI processes. Depending on the algorithm used for the collective operations, all MPI processes do not have necessarily the same amount of work. Some collectives will be very unbalanced such as `MPI_Gather` or `MPI_Bcast` when they use tree-based algorithms. In a *gather* implemented with a tree-based algorithm, the root will be receiving data on each step of the algorithm, while the leaves will only send data at operation start. These different workload will lead to different completion times on each MPI process.

Since many MPI applications are loosely based on the BSP model, or at least rely on collective communications to synchronize MPI processes, any late MPI process will delay the other ones. Thus, the overall performance depends on the slowest MPI process. We thus define the completion time of a collective as the time the last MPI process involved in this collective completes it locally.

Conversely, the start time of a collective is defined as the time where the first MPI process enters the MPI initiation call.

Hence, we define the collective global time as max(end_times) − min(start_times).

4.2 | Clock Synchronization and Barriers

To really measure the performance of the collective operations themselves, and not the drift between MPI processes, all MPI processes need to start the collective at the same time. Failing to do that leads to a phenomenon known as *process skew*[17]. Yet, synchronizing MPI processes can be tricky.

Barriers

A classical solution to synchronize the start of a benchmark is to have a barrier at the beginning of each benchmark. However, the `MPI_Barrier` itself is not unlike the other collective operations: as seen in Section 4.1, all nodes may not exit the barrier at the same time, and we cannot know how much the last rank exiting the barrier lags behind the first rank[21]. It makes the regular MPI barrier unsuitable to precisely synchronize the start of our benchmark.

Thus we implement our own barrier specifically designed to unlock all MPI processes simultaneously, using a window-based approach. The idea of such an approach is to rely on a global clock, shared between processes.

For a synchronizing window-based barrier that unlocks all process at the same time, the MPI processes do not to exit the barrier as soon as they are notified that all processes have joined; instead, they agree on a deadline in the future, in global clock, and do busy-waiting until the given deadline. This way, they all exit the barrier at the same timestamp using the global clock.

Synchronized clocks

To measure the time taken by the collective on each node, and to implement the aforementioned synchronized barrier, we need a global clock, *i.e.*, a clock that gives the same time across nodes, as much as possible. In practice, each node has its own clock. Local clocks of each node may be set to different times, and even if they are set to the same time, they may drift (the time flows at a different speed) like any clock even though they are quartz-based. This problem is usually solved at the system level using NTP[24] to synchronize clocks. However, the synchronization provided by NTP is not precise enough to measure communication times in the microsecond range. To effectively be able to synchronize all MPI processes across the multiple nodes used by the application, one must use a global clock synchronized between nodes.

We implement a global synchronized clock, taking into account both clock offset and drift using a method similar to the state of the art[20,21]. Our method uses the clock of node #0 as reference clock and computes offset and skew for the clock of each other node. At initialization time, each node go through a calibration phase where it exchange its local time with node #0 with thousands of roundtrips. We are then able to compute the offset between the local clock and the reference clock, as well as the network latency to compensate for latency in clock exchanges.

Then, to compute global clocks, we use two different methods depending on the context:

- for timestamps used only in post-mortem analysis, like the duration of collectives, we perform another calibration phase at exit, to precisely determine the offset of clocks at the end of the benchmark in addition to offsets at the beginning. Then we are able to *interpolate* each date in local time, by knowing its offset with the reference clock at the beginning and at the end, if we assume that the drift is constant.
- for timestamps that we need to translate to global clock in real time, as used for synchronizing barriers, we perform a calibration phase at the beginning of the barrier, then we *extrapolate* each local time, by knowing its offset relative to reference clock at initialization and at the beginning of the barrier, and assuming the time on each node will continue to flow at the same speed in the next seconds. To get a result precise enough, we ensure that the time between the barrier and initialization is large enough so that extrapolation does not amplify errors. We insert sleep phases if needed.

With this mechanism, we are able to synchronize all nodes with node #0. It uses a plain loop for now, but could be extended to be hierarchical[25] in future works.

4.3 | Fixed Computation and Communication Time

To thoroughly study nonblocking communications *overhead ratio* defined in Section 3.1, it is better to perform experiments with varying computation amount and communication data size, hence communication time. However, it can be very hard just looking at computation and communication sizes to assess their time, and which pair of sizes actually offer a fair matching. The time of communication depends on the number of nodes, the considered collective operation, the network hardware, just to name a few parameters. It is even worse when studying multiple MPI implementations. Each of them can use different algorithms, protocols or network interface. It can render a valid comparison point for a specific MPI implementation completely useless for another one.

Since it is not relevant to assess overlap in the case of computation time and communications time that are different by several orders of magnitude, we propose to have both scales, for computation and communication, use times rather than the number of operations for computation, and data size for communication. That way, we ensure the explored parameter space is relevant and it makes the results easier to interpret since we always know whether computation is shorter or longer than communication.

In our benchmark described in the next section, the size of data in computation and the exchanged data size in the collective are calibrated so as to reach the wanted duration.

5 | PRESENTATION OF THE BenchNBC BENCHMARK

In this section, we present BenchNBC[26], the implementation of our NBC overlap benchmark, to measure the metrics defined in Section 3 and following the methodology presented in Section 4. We also describe how each metric is presented.

5.1 | Benchmark Implementation

Estimation of Reference Times

We measure the reference communication time $t_{\text{comm\_ref}}$ first. It is defined as the duration of an MPI NBC call directly followed by an `MPI_Wait`. The wait ensures that the communication has completed. We use an NBC followed by a wait and not its blocking counterpart to be sure the same collective algorithm is used for the calibration and for the overlap benchmark, since it is not guaranteed that the MPI library uses the same algorithm for blocking and nonblocking operations. Then, as described in Section 4.3, we automatically find the data size to give to the collective so that it takes a given target time, using an iterative method and quadratic interpolation. We execute multiple runs and keep the median of all runs. Using the size and the time obtained this way, we approach the target time iteratively. We consider that this step has converged as soon as the error is below 10 %, as a compromise between precision and estimation time. It is important to note that to compute all the metrics defined in Section 3, we take for $t_{\text{comm\_ref}}$ the actual measured communication time, not the target time. In some very rare cases, this algorithm does not converge and we are unable to find a message size for the target time; in this case, the algorithm sticks to 0 bytes, and measures should actually be considered as invalid.

The second reference time needed for this benchmark is the reference computation time $t_{\text{comp\_ref}}$. The computation in the benchmark is supposed to be representative of a typical HPC application. We use a multi-threaded workload, with one OpenMP thread per core. Each thread executes a sequential GeMM (matrix multiply). We measure the time on each thread. Then, as for communication, we automatically find the matrix size so that the matrix multiplication of two square matrices takes the target time (as reference we use the time on the slowest core). Then, for $t_{\text{comp\_ref}}$ we take the actual measured computation time, not the target time.

In all of our benchmark, we did not use hyperthreading and kept one thread per physical core.

Overlap Benchmark

The overlap benchmark itself is a combination of the same code used to measure communication and computation reference times except they are now interleaved. The goal is to have both communication and computation running concurrently.

Algorithm 1: Measurement setup for overlap benchmark

```
Input: S_1  // collective data size
Input: S_2  // computation data size
t_1 ← get_Time();
MPI_Icollective(S_1);
t_2 ← get_Time();
Compute(S_2);
t_3 ← get_Time();
MPI_Wait();
t_4 ← get_Time();
```

The actual implementation is shown in Algorithm 1. Using the data and matrix size estimated with the technique described above, we first call the MPI collective function and the compute function just after. Finally, we call the `MPI_Wait` at the end of the last one.

The purpose of this test is to evaluate the ability of the runtime to make background progression. Thus, there is no call to explicit progression functions such as `MPI_Test`.

Then, using the time measured in the overlap benchmark shown in Algorithm 1, we are able to compute the inputs for our metric: $t_{\text{measured}} = t_4 - t_1$, $t_{\text{comp}} = t_3 - t_2$, $t_{\text{call}} = t_2 - t_1$, $t_{\text{wait}} = t_4 - t_3$.

The benchmark is designed to run with 1 MPI process per node, with all cores occupied with OpenMP threads. It is possible to reduce the number of OpenMP threads to leave one (or several) cores free for any MPI progress thread.

5.2 | Metrics Display

We display the metrics described in Section 3 with different methods.

Overhead ratio display

The main metric we show is the *overhead ratio* r_{overhead} as defined in Section 3.1. We represent it using a 2-D heat-map, with communication time as X-axis and computation time as Y-axis. The color map ranges from green ($r_{\text{overhead}} = 0$, perfect overlap) to yellow ($r_{\text{overhead}} = 1$, serialized execution), and further to red ($r_{\text{overhead}} > 1$, slowdown). Blue means $r_{\text{overhead}} < 0$ (faster than reference time), which usually indicates measure imprecision[2].

X-axis and Y-axis follow the same scale, hence the bottom left to top right diagonal represents cases where computation and communication times are equal. In the opposite corners, the two timings differ from several orders of magnitude (500 μs v.s. 1 s). Hence, the measures and their ratios are very sensible to execution noise, which leads to less meaningful observations. We did not crop them on the figures for completeness, but they may be safely ignored when looking at the 2-D heat-maps.

Impact on computation display

The impact on computation, $r_{\text{MPI\_impact}}$ is shown as a histogram, as in Figure 9; the red line highlights a ratio of 1 (no impact).

Concurrency ratios display

Concurrency ratios, r_{comm} and $r_{\text{comp\_slowdown}}$, are also presented as 2-D heat-maps, with the same axes as for the *overhead ratio*. The color map ranges from blue for the best case (for communication ratio, perfect overlap with $r_{\text{comm}} = 0\%$; for computation ratio, no impact with $r_{\text{comp\_slowdown}} = 100\%$) to red for the worst case (communication slowdown with $r_{\text{comm}} > 100\%$; impact on computation with $r_{\text{comp\_slowdown}} > 100\%$). We show these ratios in pairs, with communication ratio heat-map on the left, and computation ratio heat-map on the right, as in Figure 7.

5.3 | Experimental sSetups

We present here our experimental setups for the tests with our benchmark. We run experiments on various MPI implementations and network hardware. Our test platforms are:

- *inti/skylake*: 32 nodes, with dual-socket Intel Xeon Platinum 8168, each with 24 cores at 2.7 GHz, equipped with Mellanox MT27700 (Connect-IB) InfiniBand boards;
- *inti/haswell*: 8 nodes, with dual-socket Intel Xeon E5-2698, each with 16 cores at 2.3 GHz, equipped with Mellanox MT27600 (Connect-IB) InfiniBand boards;
- *inti/sandy*: 64 nodes with dual-socket Intel Xeon E5-2680, each with 8 cores at 2.7 GHz, equipped with Mellanox MT25400 (ConnectX-2) InfiniBand boards;
- *irene/bxi*: 64 nodes, with Intel Xeon Phi 7250 with 68 cores at 1.4 GHz, equipped with Bull BXI v1.2 network adapters;
- *bora/omnipath*: 8 nodes with dual-socket Xeon Gold 6240 each with 18 cores at 2.60 GHz, equipped with Omni-Path HFI Silicon 100 Series network adapters.

We run one MPI process per node, one thread per physical core, without hyperthreading. We tested four different collectives for each configuration: `MPI_Ibcast`, `MPI_Ireduce`, `MPI_Iallgather` and `MPI_Ialltoall`.

6 | BENCHMARK CASES STUDY

In this section, we present some benchmark results in two typical cases: OpenMPI/InfiniBand in *basic* configuration and MPICH with *async progress*; and detail how to use our metrics to diagnose pathological behaviors. We then compare these results and the approach of our benchmark with state-of-the-art benchmarks IMB[15] and OSU[16] by applying these benchmarks to the same cases.

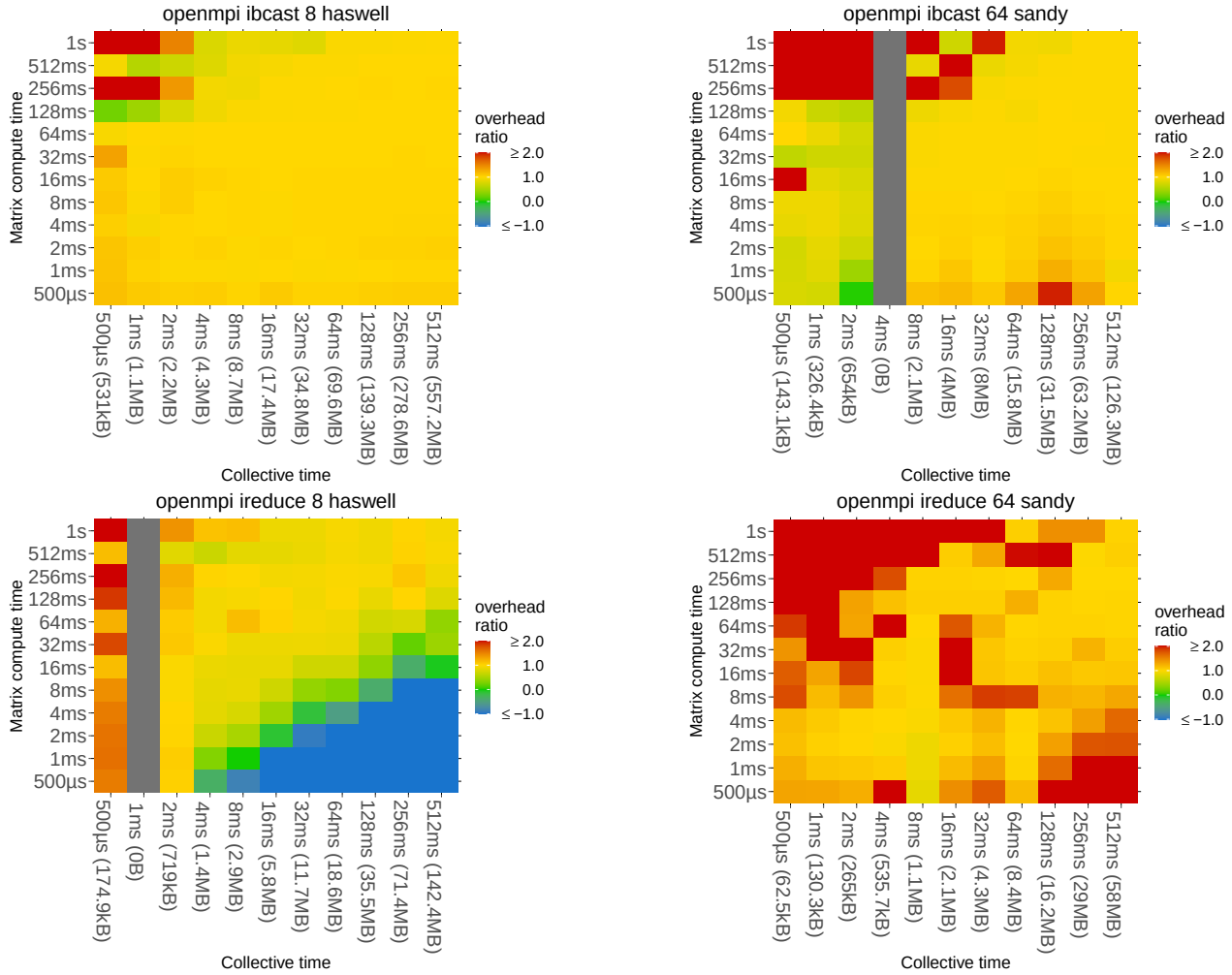

FIGURE 6 OpenMPI 4.0.2 overlap results on InfiniBand for `MPI_Ibcast` and `MPI_Ireduce` on 8 nodes (left) and 64 nodes (right)

6.1 | Case sStudy: OpenMPI/InfiniBand

We present here results for the specific case of OpenMPI 4.0.2 on the two machines *inti/haswell* (using 8 nodes) and *inti/sandy* (using 64 nodes), both with an InfiniBand network.

Figure 6 shows *overhead ratio* results for `MPI_Ibcast` and `MPI_Ireduce`. In most cases, we observe no overlap at all (yellow areas), with $r_{\text{overhead}} \approx 1$, and even *slowdown* in some cases (red areas, with $r_{\text{overhead}} > 1$. This poor performance may be surprising for some users, but it gets easily explained: OpenMPI does not feature mechanisms for asynchronous progression. It is expected to observe no overlap in case there is no background progression mechanisms in the MPI library, like described in Section 3.3.1.

To explain the lack of overlap, Figure 7 shows concurrency ratios r_{comm} and $r_{\text{comp\_slowdown}}$ for OpenMPI on `MPI_Ibcast` and `MPI_Ireduce`. `MPI_Ibcast` on 8 *inti/haswell* nodes exhibits an r_{comm} typical of a purely sequential behavior: MPI calls are as long as without overlap, which shows it was not executed in background; the computation is unaffected. For `MPI_Ireduce`, r_{comm} (left) is red, indicating that the communication is slowed down by the computation, while the computation (right) is not impacted. The difference between `MPI_Ireduce` and `MPI_Ibcast` is that `MPI_Ireduce` needs to execute the reduction operation

[2]Warning to Mac users. The Apple Quartz engine tends to blur the figures displayed in this section, even for printing. To correctly view and print the figures, use a non-Quartz PDF renderer (*e.g.*, Adobe or Chrome).

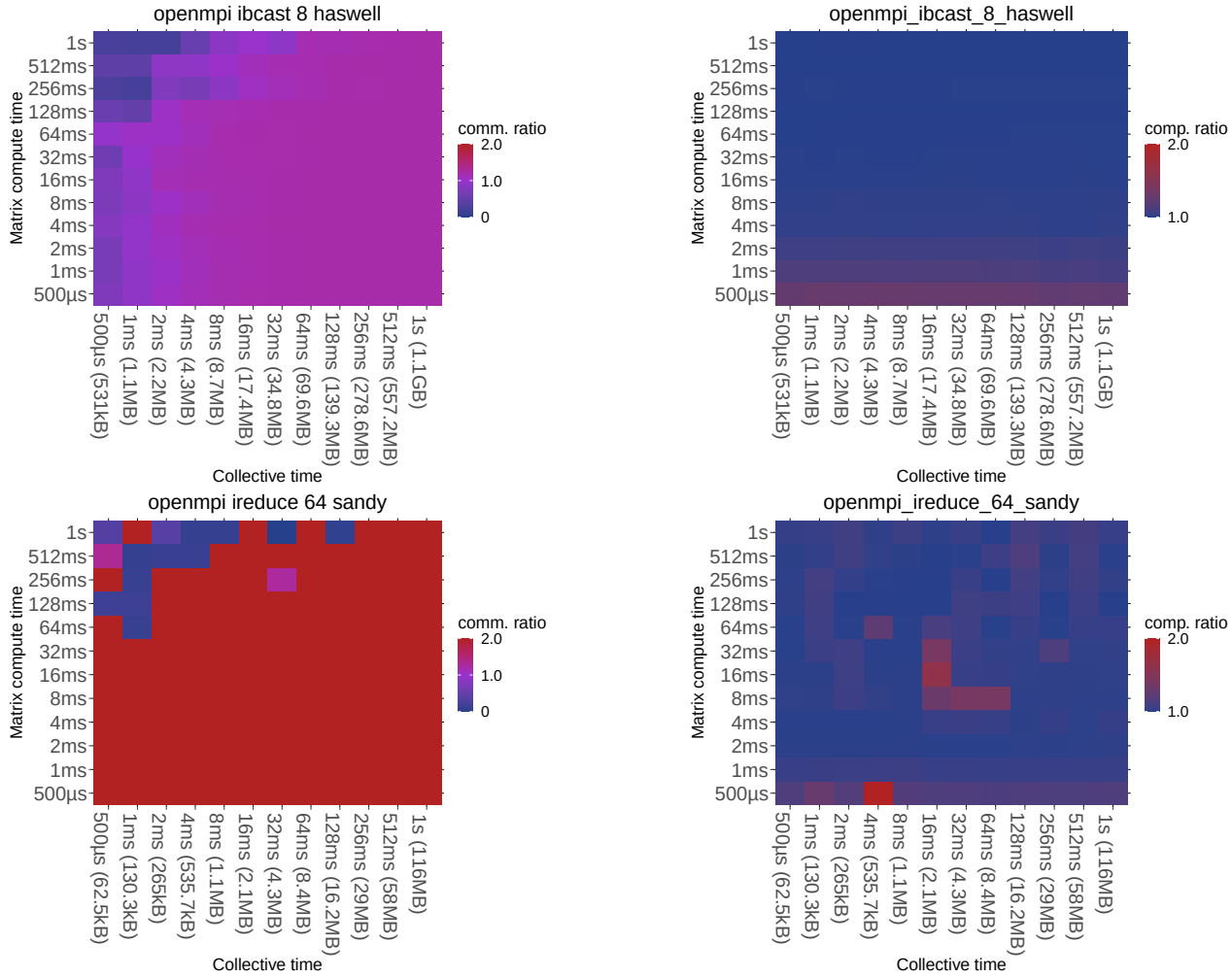

FIGURE 7 OpenMPI concurrency ratios r_{comm} (left) and $r_{\text{comp\_slowdown}}$ (right), on 8 nodes (top) and 64 nodes (bottom)

on the CPU; these results show that the reduction speed actually suffers from being run alongside computation. `MPI_Ireduce` on 64 nodes is the case where the communication time is the most impacted.

Moreover, we observe some green areas with $r_{\text{overhead}} < 1$ (or even blue) for `MPI_Ireduce` on *inti/haswell* on Figure 6, which indicates successful overlap in this part of the parameters space. The green area is located below the diagonal, thus with computation time larger than communication time. We can guess that a single step of the collective algorithm actually overlaps communications with computation, thanks to hardware progression for point-to-point operations, then the remainder of the collective does not have the opportunity to be scheduled on the CPU while the computation is running and is scheduled only at the end. Such mechanism only works when the computation is shorter than a single step of the collective algorithm, so much shorter than the full collective time (computation 4 times shorter than communication, which corresponds to a single level in a reduction tree on 8 nodes). Obviously, the more nodes, the more insignificant this phenomenon becomes.

As a conclusion, in most cases no overlap is observed with OpenMPI, which is not surprising given that it does not implement asynchronous progression mechanisms. We have shown that there is overlap in some anecdotal cases with very short computation and a low nodes count. We have exhibited pathological cases (*e.g.*, `MPI_Ireduce`) where the communication is *slower* than if no overlap would have been attempted. In the general case, the observed performance is roughly the same as if computation and communication would have been run sequentially.

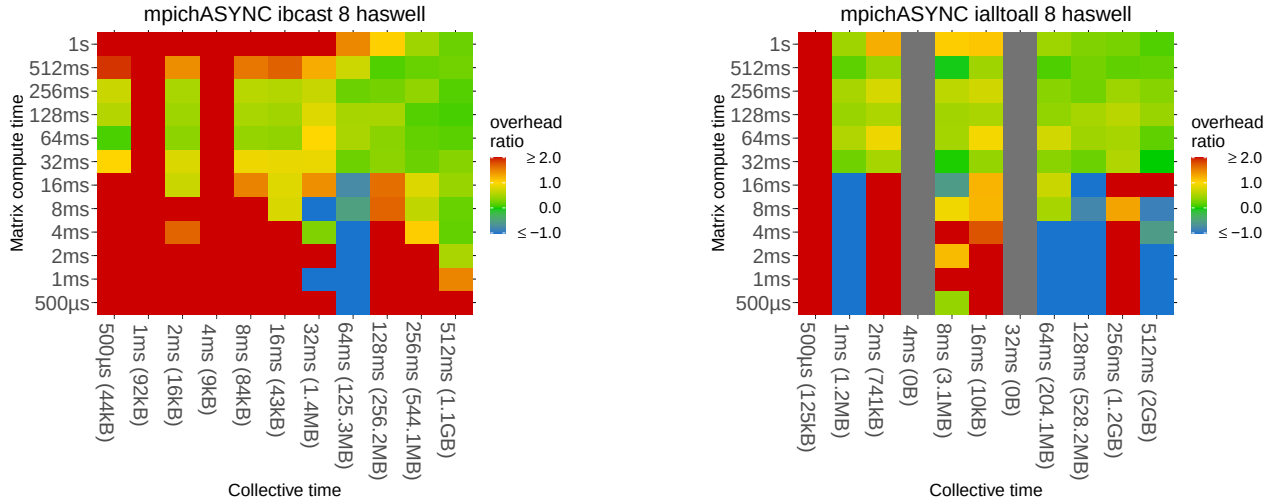

FIGURE 8 MPICH overlap results with async progress thread on 8 nodes

MPI library	madmpi	madmpi1DED	mpich	mpichASYNC
$r_{\text{MPI\_impact}}$	1.01	1.00	1.00	1.27

FIGURE 9 Runtime impact comparison on 512 ms computation time, for MadMPI (left 2 columns - dedicated core and no dedicated core) and MPICH (right 2 columns - async progress enabled and disabled)

6.2 | Case sStudy: MPICH with `MPICH_ASYNC_PROGRESS=1`

MPICH features an optional progress thread that can be activated through the `MPICH_ASYNC_PROGRESS` environment variable. Results for the overhead ratio are shown on Figure 8. As expected, we observe successful overlap in some cases. However, it only works for large compute sizes and large communication sizes. With smaller sizes, there is a significant slowdown, worse than without the progress threads. We observe intermixed zones with $r_{\text{overhead}} \gg 1$ and zones with $r_{\text{overhead}} < 0$. This is due to huge variability of performance, with standard deviation between 4 ms and 11 ms. When looking at the message sizes found by our calibration method for each collective time, we can see that it is non-monotonic. Actually, the behavior looks like it is *chaotic*. For larger sizes and compute times long enough, these variations become negligible and overlap can be seen. For small sizes, these variations prevent from having meaningful times and observations. We assume that the high variability of performance comes from perturbations from the MPI runtime with its progression thread.

To confirm this hypothesis, we measured the impact of the MPI implementation on the computation, using the $r_{\text{MPI\_impact}}$ introduced in Section 3.2.1. The ratio is shown in Figure 9 for estimated compute times of 512 ms, on the right two bars for MPICH with asynchronous progression and without progression. Overall, for all estimated compute times, MPICH with an asynchronous progress thread has a huge impact on the computation, even without actually doing any MPI communications. It causes a computation slowdown up to 50 % in the worst case. This is explained by the progress thread being scheduled by the kernel on cores performing computation, stealing CPU cycles and thus delaying computation. This behavior has not been observed with MadMPI nor with MPICH without asynchronous progression, which gets $r_{\text{MPI\_impact}} \approx 1$.

To understand the impact of MPICH progress thread in the case of overlap, we take a look at the concurrency ratios r_{comm} and $r_{\text{comp\_slowdown}}$, as introduced in Section 3.2.2, for the `MPI_Ibcast` case, in Figure 10. We observe that the bad overhead ratio observed for small compute sizes and short collective communication times comes from two different reasons. For small compute times, the overhead comes from the impact on computation. It confirms that the MPICH runtime in this configuration has a huge impact on the computation. Hence, the slowdown induced on the computation jeopardizes the global performance, even with perfect overlap. For communications, above a 32 ms threshold, we observe that the larger the times, the smaller the communication time spent in the `MPI_Wait` function, until the communication time is completely overlapped by the computation. For small communication sizes, the observed overhead comes from communication slowdown. In these cases, the extra cost

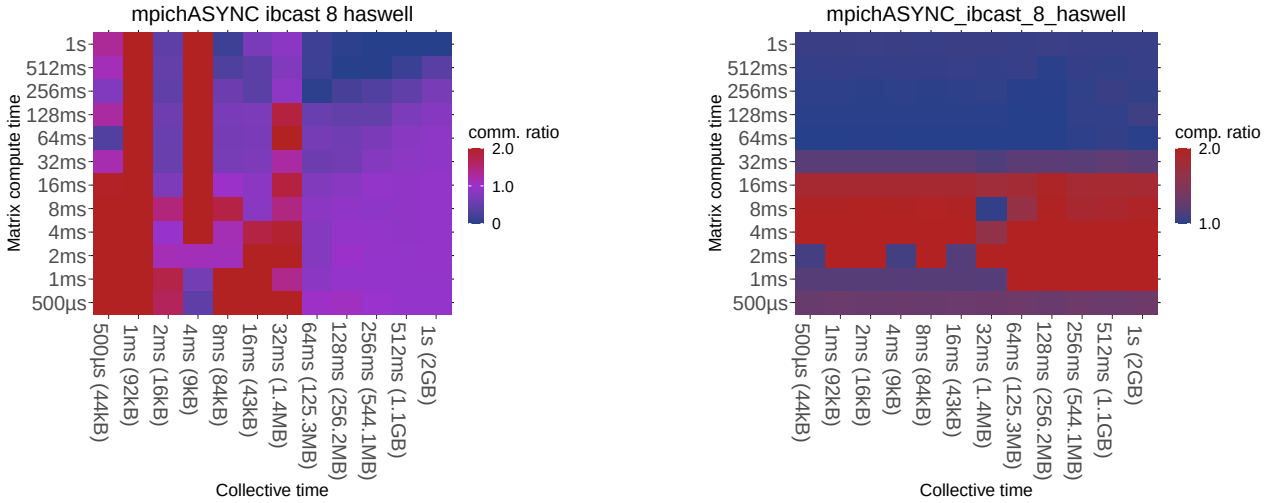

FIGURE 10 MPICH concurrency ratios with async progress thread r_{comm} (left) and $r_{\text{comp\_slowdown}}$ (right) on 8 nodes

```
OSU MPI Non-Blocking Broadcast Latency Test v5.7
# Size            Overall(us)       Compute(us)    Pure Comm.(us)      Overlap(%)
 1048576                665.10            424.78           412.18           41.70

# Size            Overall(us)       Compute(us)    Pure Comm.(us)      Overlap(%)
 1048576               1226.44            706.79           681.85           23.79
```

FIGURE 11 OSU benchmark results for 1MB `MPI_Ibcast` and `MPI_Ireduce` with OpenMPI on *inti/haswell* (using 8 nodes)

necessary to use a progress thread is too high to be hidden by the communication overlap gain when CPU computations are required.

6.3 | Comparison with sState-of-the-aArt

We compare here our metrics and benchmark with state-of-the-art benchmarks for nonblocking collectives: IMB [15] and OMB (OSU microbenchmark) [16]. To do so, we compare the results on both cases studied in Sections 6.1 and 6.2.

General idea.

The general principle of both OMB and IMB is the use of an *overlap ratio*, defined as the ratio of time spent in the MPI primitives with overlap, relative to the same time with blocking MPI calls. Overlap is done with a computation running approximately the same time as the reference communication time. This way, they evaluate how much time spent in the MPI primitives has decreased. In essence, this is very similar to our r_{comm} metric.

In contrast, the *overhead ratio*, our main metric, compares what we get on the overall time with what we expected if perfect overlap would happen. As a consequence, our benchmark captures behaviors where the progression mechanisms interfere with computations and make them less efficient.

Comparison of benchmarks using OpenMPI/InfiniBand.

The first case for our comparison is OpenMPI as described in Section 6.1. The results obtained with OSU benchmark v5.7 for 1MB `MPI_Ibcast` and `MPI_Ireduce` are depicted in Figure 11. It shows a 41.70% overlap ratio for the broadcast and 23.79% for the reduction. In comparison, our results for the same reduction case shown in Figure 6 are in 2-D, with computation time $t_{\text{comp\_ref}}$ and communication time $t_{\text{comm\_ref}}$ not necessary equal. We observe on our graphs that results are non-uniform, thus it is valuable to measure overlap for a computation time different than the communication time. In practice, an application programmer tries to overlap communications with computation, but he/she has no guarantee their duration will be equal, depending on the CPU

```
OSU MPI Non-Blocking Broadcast Latency Test v5.7
# Size           Overall(us)       Compute(us)    Pure Comm.(us)     Overlap(%)
 1048576            2098.49           1888.42          1832.79          88.54

# Size           Overall(us)       Compute(us)    Pure Comm.(us)     Overlap(%)
 1048576             851.56            663.64             643.58          70.80
```

FIGURE 12 OSU benchmark results for 1MB `MPI_Ialltoall` and `MPI_Ireduce` with MPICH with asynchronous progression on *inti/haswell* (using 8 nodes)

```
#    Intel(R) MPI Benchmarks 2018, MPI-NBC part
#bytes    #repet.    t_ovrl[usec]   t_pure[usec]  t_CPU[usec]   overlap[%]
 1048576       40         2281.20         1888.82     2052.86          80.89

#bytes    #repet.    t_ovrl[usec]   t_pure[usec]  t_CPU[usec]   overlap[%]
 1048576       40           981.55          790.55        803.90         76.24
```

FIGURE 13 IMB benchmark results for 1MB `MPI_Ialltoall` and `MPI_Ireduce` with MPICH with asynchronous progression on *inti/haswell* (using 8 nodes)

speed, network speed, and number of nodes, performance is hard to predict. Our results cover a wider range of values, though the conclusion is not radically different: overlap is not very good in this case.

Because we display various computation time and communication time, we can observe that a better *overhead ratio* is available for the reduction case (middle left 2-D heat-map in Figure 6, labeled *openmpi ireduce 8 haswell*). If the communication time is greater than the computation, then we have a greener diagonal, indicating a better overlap. This is easily explained thanks to the two *concurrency ratios*, r_{comm} and $r_{\text{comp\_slowdown}}$, as described in Section 6.1. In the bottom 2-D heat-maps of Figure 7, r_{comm} (left) is red, indicating that the communication is slowed down by the computation, while the computation (right) is not impacted. Hence, to have a perfect overlap, the *slowed* communication should not take more time than the computation. Because the slowed communication takes twice (or more) the amount of time of the communication without computation, having an initial communication time smaller than the computation time will bring a better overlap. If a user can influence the nonblocking communication/computation time ratio in its program, having such information can help leverage better performance.

Such information cannot be deduced from the IMB or OSU measurements, but only with our extended metrics.

Comparison of benchmarks using MPICH + asynchronous progression.

The second case is MPICH with asynchronous progress thread. We consider `MPI_Ialltoall` and `MPI_Ireduce`, with 1MB of data, on 8 nodes of the *inti/skylake* platform. Our results are described in Section 6.2 (Figure 8), and OSU benchmark and IMB results are shown in Figure 12 and Figure 13 respectively.

We can see that there is a huge difference between OSU/IMB-NBC and our benchmark: our benchmark shows that overlap causes an overall 10× *slow down* ($r_{\text{overhead}} = 10.96$ for `MPI_Ialltoall` and 13.07 for `MPI_Ireduce`, on the considered data size), while other benchmarks tend to show it overlaps successfully ($overlap = 70.80\%$ to 88.54%). It is explained by multiple factors:

- nature of metric: our metric, which is an *overhead* relative to perfect overlap, is an open scale; it can express the fact that an overlap leads to worse performance than if no overlap would have been attempted.
- realistic computation: the computation method in our benchmark uses OpenMP, and thus exploits all cores, compared to the single-threaded computation in other benchmarks. We believe our approach is closer to real applications, and is able to show thread interaction (applications threads and MPI progress threads) that other benchmarks would overlook.
- computation fixed time v.s. fixed amount: our benchmark runs a predefined amount of work, so as to be able to compare the duration of computation with and without overlap, and thus be able to detect the impact of overlap on computation speed. For computation, the other benchmarks use a loop that runs for a given time, without being able to tell whether the

amount of computation was hindered by communication progression mechanisms. Our metric $r_{\text{comp\_slowdown}}$ is designed to pinpoint this issue.

- timing issues: other benchmarks take the average of time between nodes as the collective time, while we take the time of the last rank to complete the collective. It is important especially for collectives implemented as trees, with a huge load-imbalance between nodes. Moreover, thanks to our synchronized clocks, we take the median time of a number of iterations, while others use simply the average.

Our interpretation on this MPICH +progression case, as explained in Section 7.2.1, is that activating the progress thread makes it collide with OpenMP application threads, causing a huge overhead: the progress thread needs to be woken up, then it competes with application threads, leading to slow computation, and slow communication. We observe that this slowed-down communication is actually overlapped, even though this is probably not what the user wants. All these phenomenons are overlooked by other benchmarks that simply conclude that there is overlap.

Main differences.

More generally, IMB and OSU benchmarks try to measure whether there is overlap or not in the case of perfectly matching computation and communication time. Our benchmark consider more realistic cases:

- it uses parallel computation with OpenMP, closer to nowadays applications;
- it assess not only the speedup, but also the slowdown on the communication time, if any;
- it tests various computation and communication times, allowing to find the best combination for overlap.

Our benchmark tries not only to assess overlapping but to diagnose pathological behaviors. It may be useful for MPI library developers, but for end-users too, to diagnose thread conflicts between the MPI library and the application.

7 | SURVEY OF OVERLAP BENCHMARK RESULTS WITH VARIOUS MPI IMPLEMENTATIONS AND NETWORKS

We present here a survey of the results obtained with our benchmarks on a large set of configurations and MPI libraries described in Section 5.3. See[26] for more complete results.

We distinguish three different deployment configurations:

- *basic* with computation on all cores, and default configuration of MPI library;
- *progress* with computation on all cores, and asynchronous progression mechanisms enabled if available;
- *dedicated* with computation on $n - 1$ cores, with progress thread enabled and bound to the free core if possible.

7.1 | Results with ~~b~~Basic eConfiguration

We present here results for the *overhead ratio* we get with widespread MPI implementations and their default configuration. Due to space constraints, we cannot include results for all MPI implementations, on all machines, for all collective operations. The selected results are typical of what we obtained.

These results are obtained on InfiniBand network, on machines *inti/haswell* and *inti/sandy*. Figure 14 shows results for Open-MPI 4.0.2, in addition to Figure 6 already studied in Section 6.1, Figure 15 for MVAPICH 3.2.1, Figure 16 for MPICH 3.3, and Figure 17 for MPC.

We also present results obtained for OpenMPI with TCP-Ethernet on *inti/sandy* on Figure 18, and with Omni-Path on *bora/omnipath* on Figure 19.

Since we are in the basic configuration here, the MPI implementations do not feature mechanisms for asynchronous progression. Most of these results are similar to the case studied in Section 6.1: no progression is observed and communication and computations are serialized (yellow areas), or they are contending with each other so the overall performance is slower then serialized (red areas). We get the same green areas with $r_{\text{overhead}} < 1$, which is a sign of successful overlap, in the bottom right portion of graphs, where computation is much shorter than communication.

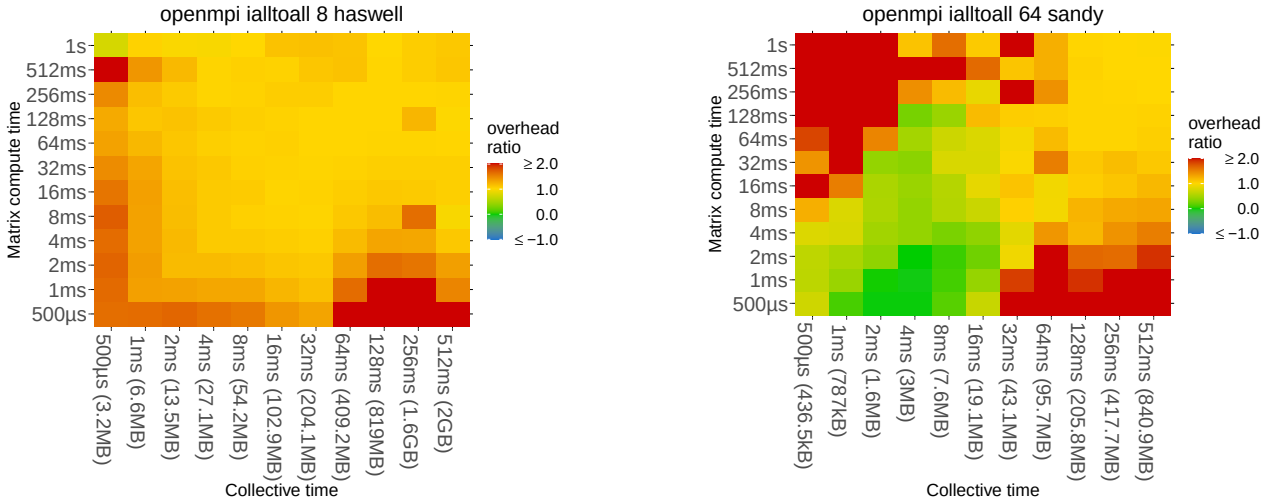

FIGURE 14 OpenMPI 4.0.2 overlap results on InfiniBand for `MPI_Ialltoall` on 8 nodes (left) and 64 nodes (right)

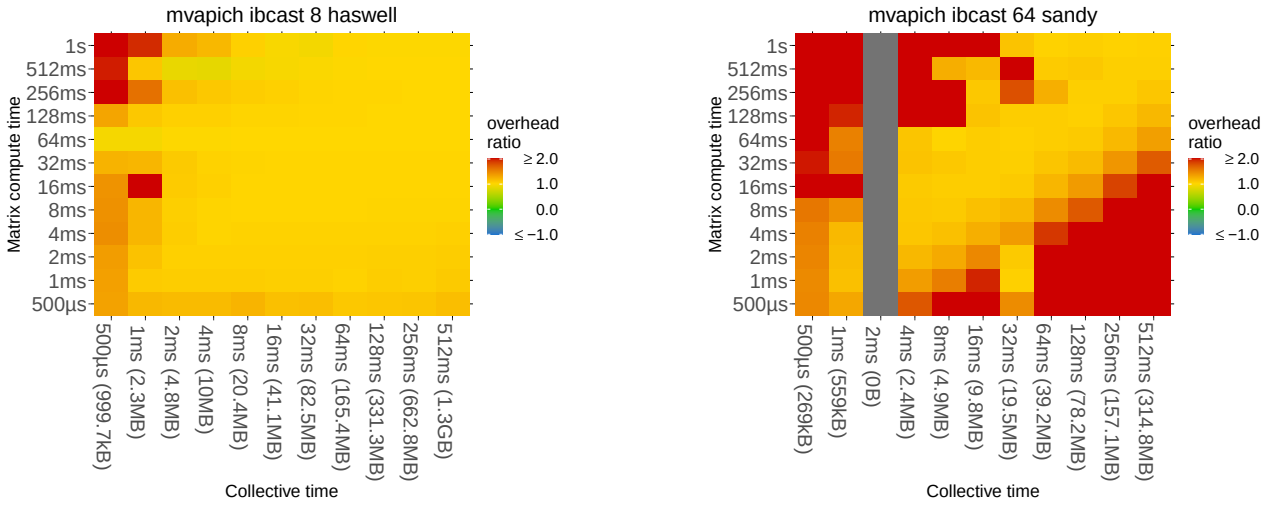

FIGURE 15 MVAPICH 3.2.1 overlap results on InfiniBand for `MPI_Ibcast` on 8 nodes (left) and 64 nodes (right)

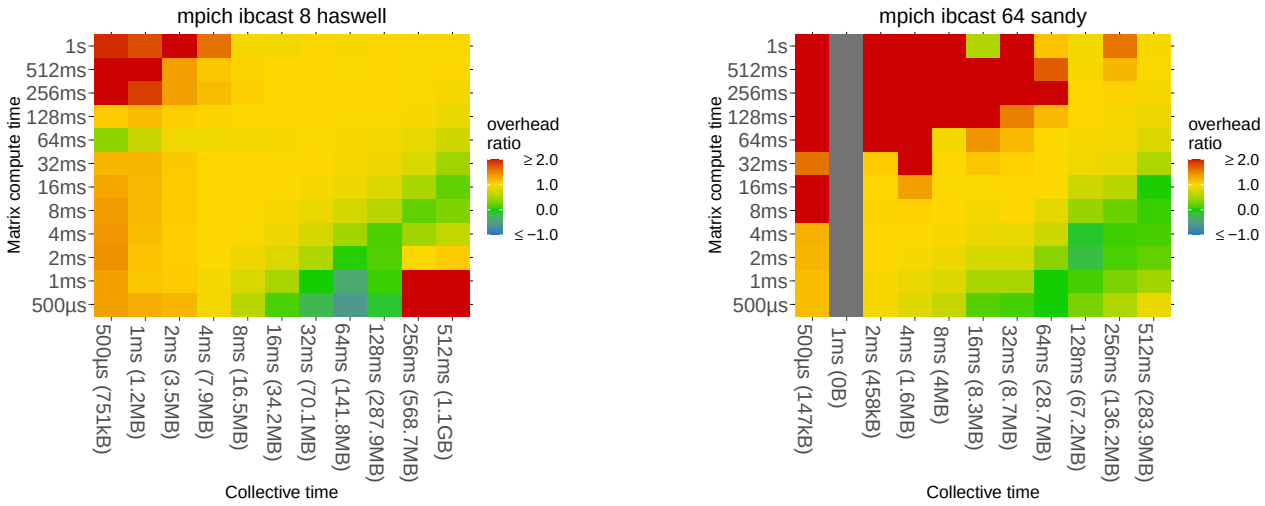

FIGURE 16 MPICH 3.3 overlap results on InfiniBand for `MPI_Ibcast` on 8 nodes (left) and 64 nodes (right)

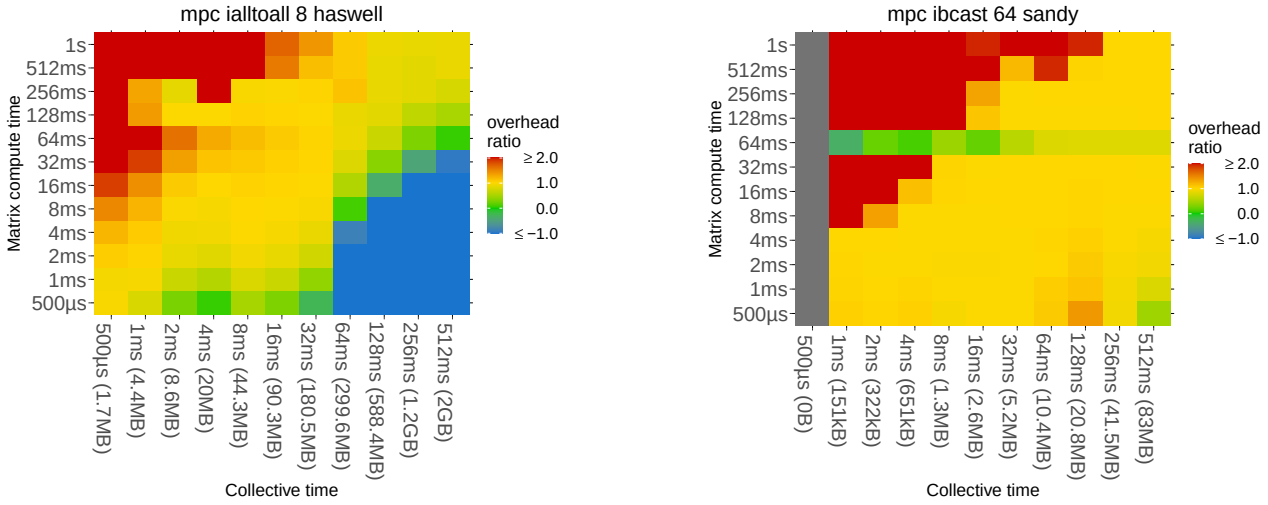

FIGURE 17 MPC overlap results on InfiniBand for `MPI_Ialltoall` and `MPI_Ibcast` on 8 nodes (left) and 64 nodes (right)

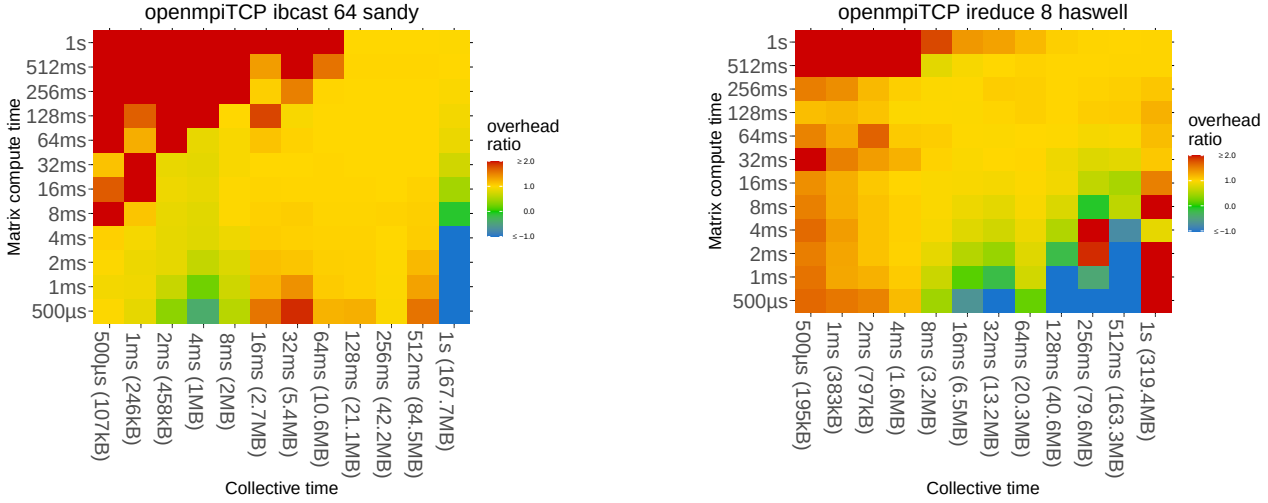

FIGURE 18 OpenMPI 4.0.2 overlap results on TCP-Ethernet for `MPI_Ibcast` on 64 nodes and `MPI_Ireduce` on 8 nodes

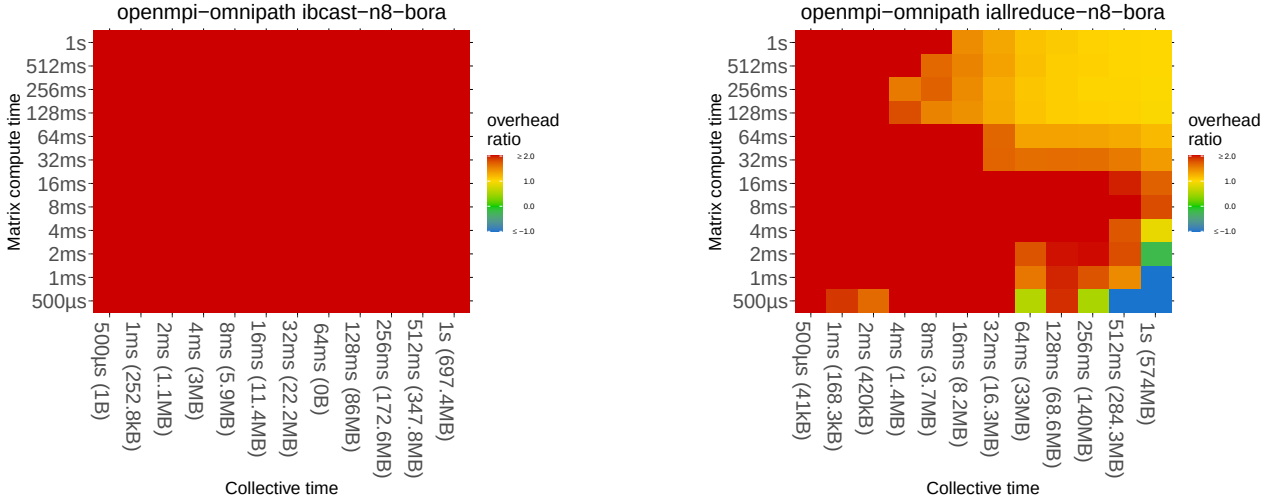

FIGURE 19 OpenMPI 4.0.2 overlap results on Omni-Path for `MPI_Ibcast` and `MPI_Iallreduce` on 8 nodes

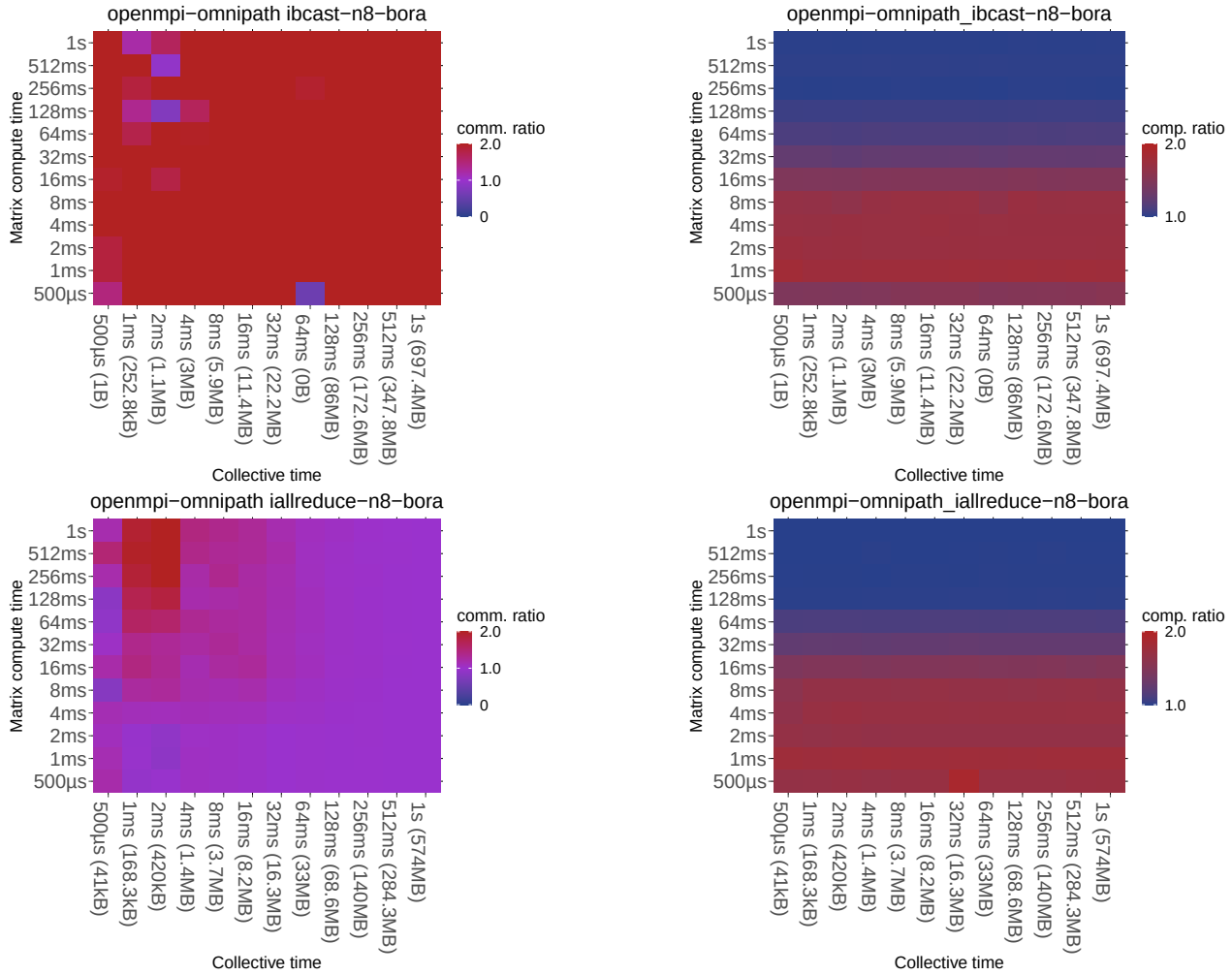

FIGURE 20 OpenMPI concurrency ratios r_{comm} (left) and $r_{\text{comp\_slowdown}}$ (right) on Omni-Path on 8 nodes

We observe another green area for OpenMPI on small communication and short computation for `MPI_Ialltoall` on sandy (Figure 14), probably because without *rendezvous* and without dependency between messages as in `MPI_Ialltoall`, progression is performed fully by the hardware as independent point-to-point operations.

The overlap results for Omni-Path are worse than the other OpenMPI results. The corresponding concurrency ratios in Figure 20 show that both communication and computation are impacted on this machine, jeopardizing any chance of performance improvement through overlap.

As a conclusion, with basic configuration, for a large set of MPI libraries (OpenMPI, MVAPICH, MPICH, MPC), on various networks (InfiniBand, TCP-Ethernet, Omni-Path), no overlap is observed except in some insignificant cases. We even observe performance degradation in some cases.

7.2 | Results with aAsynchronous pProgression eEnabled

We present here results with asynchronous progression enabled.

7.2.1 | MPICH with `MPICH_ASYNC_PROGRESS=1`

MPICH with `MPICH_ASYNC_PROGRESS=1` has been detailed previously in Section 6.2. In some cases, overlap is successful; in other cases, it causes contention between the progress thread and computation, which causes a huge performance drop.

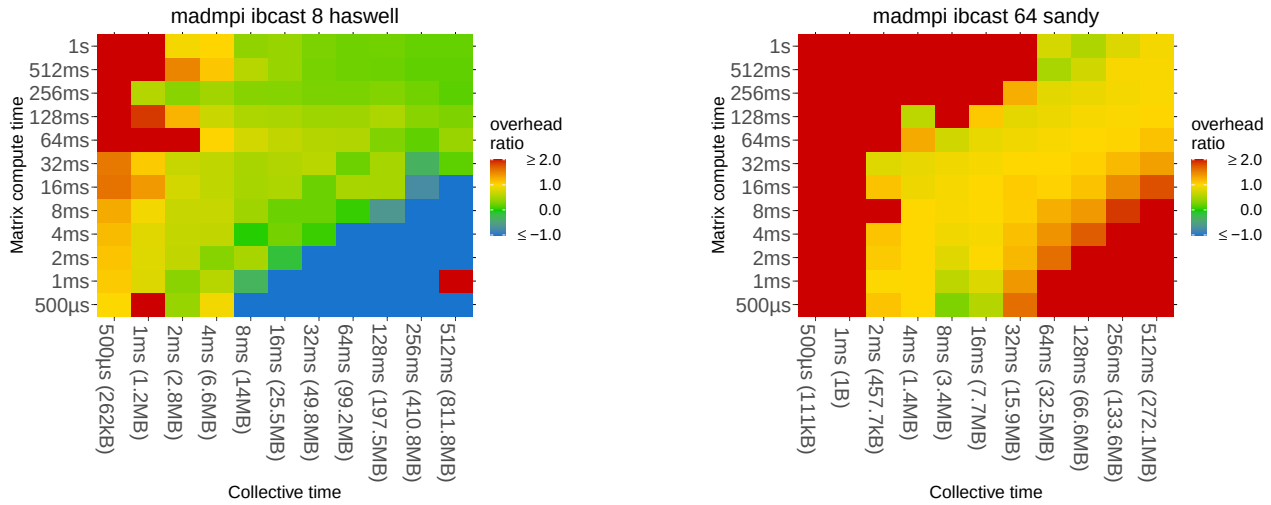

FIGURE 21 MadMPI overlap results with default parameters on InfiniBand on 8 nodes (left) and 64 nodes (right)

7.2.2 | MVAPICH on InfiniBand

MVAPICH also provides a version supporting progress threads: MVAPICH2-X. Unfortunately, it is distributed as binary-only and we were unable to correctly install and use MVAPICH2-X 3.2 on our test machines, thus we cannot provide any result with it.

7.2.3 | MPC

Recently, MPC exhibited good performances when activating its progress thread[8]. However, this support appears to be broken in the tested version, thus no results can be provided.

7.2.4 | MadMPI with dDefault pProgression

MadMPI enables progression mechanisms[7] by default. Figure 21 shows its overhead ratio for `MPI_Ibcast`. We can observe successful overlap on 8 nodes, but no overlap or even slowdown on 64 nodes.

Figure 9 (in Section 6.2) shows that MadMPI progression mechanism has nearly no impact on computation when no communications occur, since its background progression mechanisms are designed to have more gentle polling strategy than busy-waiting. Progression mechanisms in MadMPI were initially designed to handle nonblocking point-to-point. The workload of collective being heavier, sometimes its progression mechanisms are not enough to achieve good overlap.

Figure 22 displays the concurrency ratios for `MPI_Ibcast` on 64 nodes. We observe on these figures that the reason for the bad overhead ratio for this case is twofold. First, with a r_{comm} around 100% for the most part, it seems communications are not overlapped at all. For the cases where communications are overlapped, the purplish spots for $r_{\text{comp\_slowdown}}$ indicates computation is impacted. Since on this part of the figure, computation time is larger than communication times, no performance gain is visible.

This behavior may be explained by the fact that in this configuration, progression is performed by a thread with uncontrolled placement, thus colliding with computation threads.

7.3 | Results with dDedicated eCore for pProgression

We present here results with a core dedicated to communication progression in the libraries that are able to do so. In this case, computation uses all cores but one. The workload *per core* is the same as without a dedicated core.

7.3.1 | MPICH on InfiniBand with dDedicated eCore

We tried to dedicate a core to MPI progression with MPICH, by running computation on all cores except one, and enabling `MPICH_ASYNC_PROGRESS=1`. Unfortunately, we found no mechanism to bind the progress thread to the dedicated core, so its

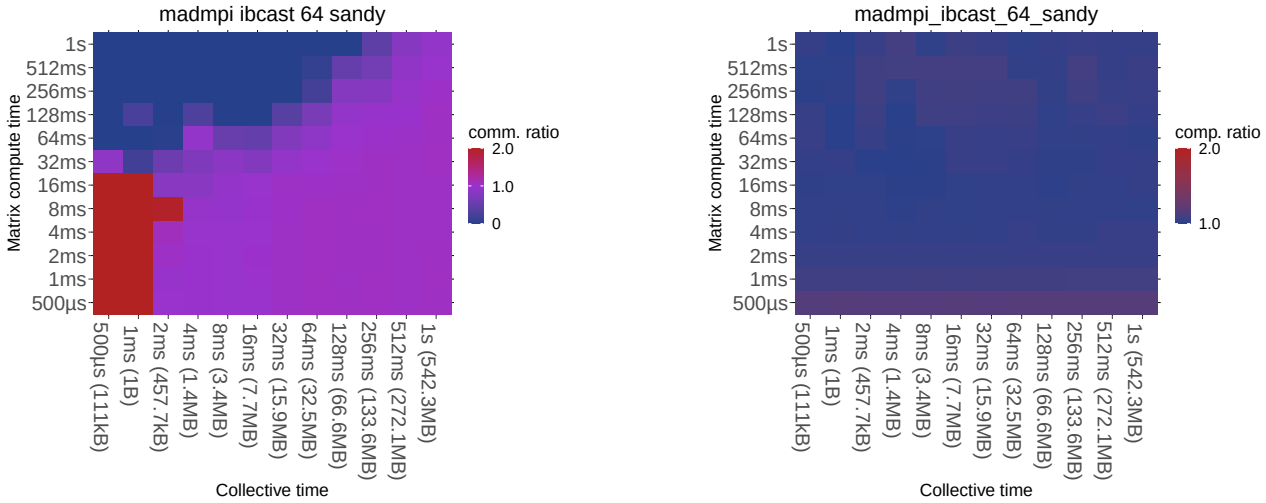

FIGURE 22 MadMPI concurrency ratios on InfiniBand on 64 nodes

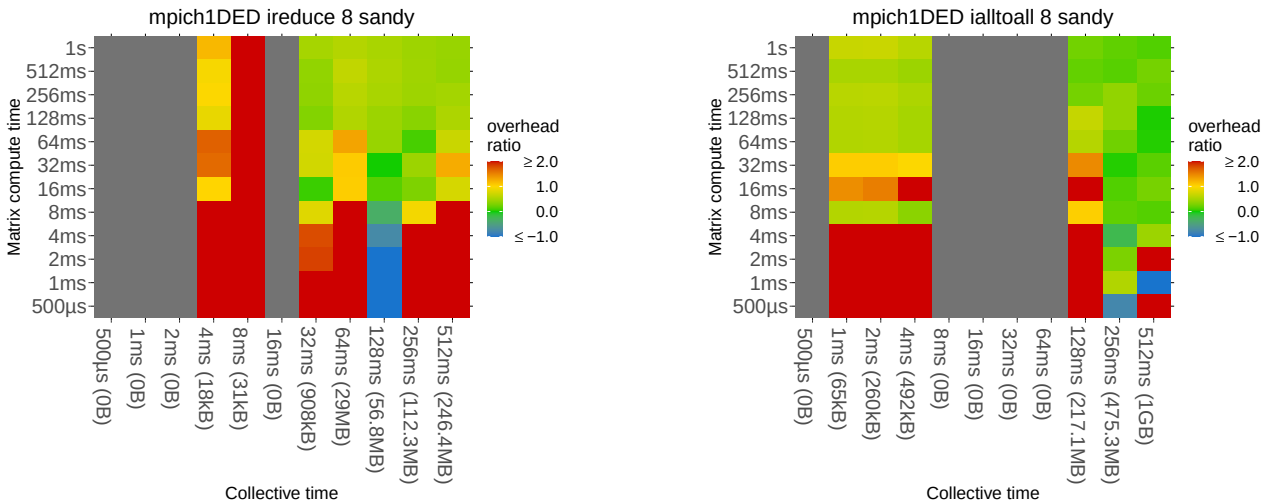

FIGURE 23 MPICH overlap results with dedicated core on InfiniBand on 8 nodes

placement was handled by the kernel thread scheduler. The observed overhead ratios are shown in Figure 23. They are very similar to the ones depicted in Figure 8 without the dedicated core.

7.3.2 | MadMPI with dDedicated eCore

We enabled the dedicated core configuration in MadMPI and reduced the number of OpenMP threads by one. Overhead ratio of this configuration is shown in Figure 24. We observe that MadMPI with a dedicated core manages to overlap perfectly communications with computation, for any computation/communication sizes. The efficient overlap also scales up to the tested 64 nodes, though the computation and communication times need to be closer than on 8 nodes.

Figure 9 shows $r_{\text{MPI\_impact}}$, the impact of MadMPI on computation (left 2 bars), with the dedicated core and without the dedicated core. The default has little impact on computation but the configuration with dedicated core reduces this impact, which is not surprising considering that the default uses the same cores as computation, whereas having a dedicated core should avoid any interaction.

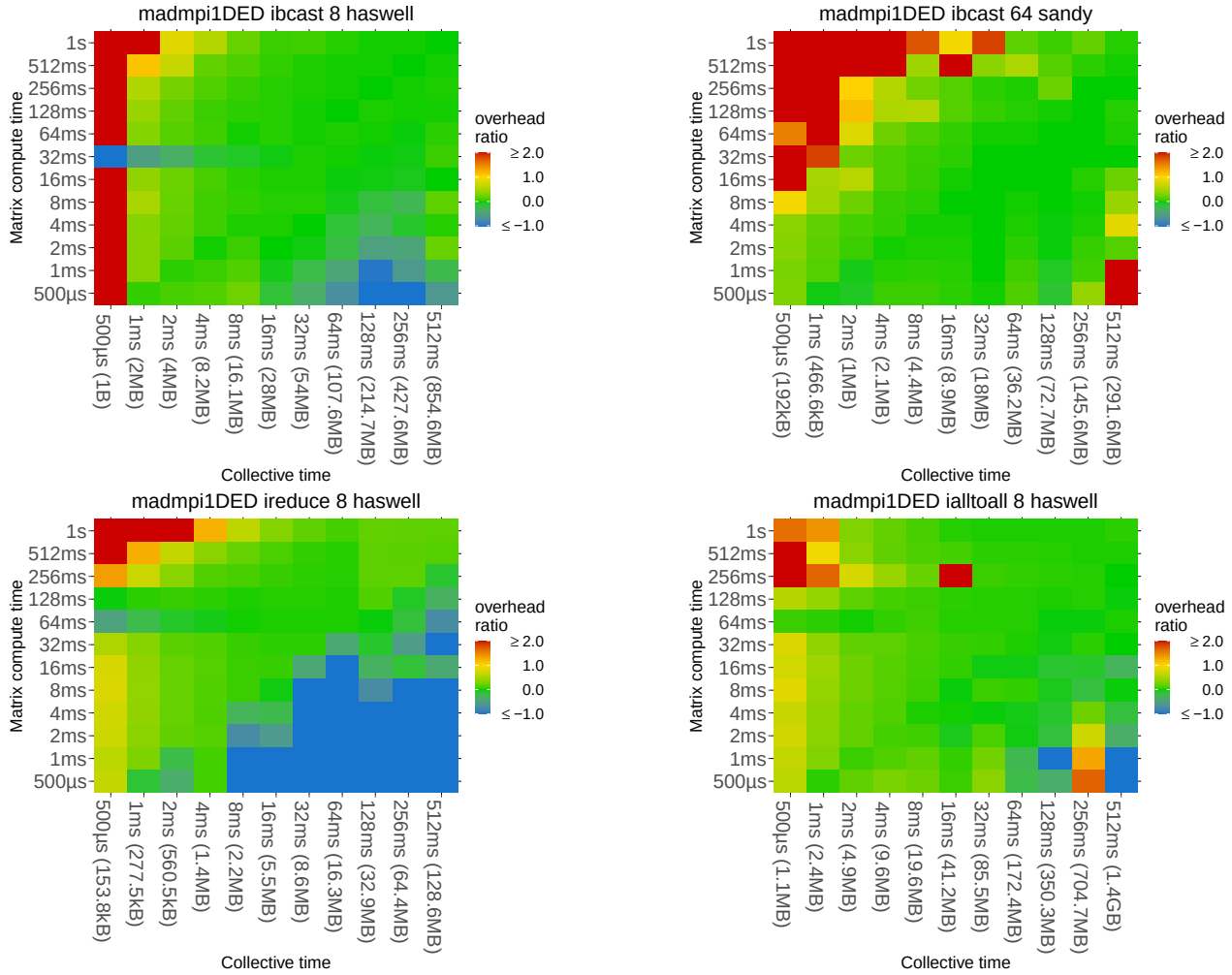

FIGURE 24 MadMPI overlap with dedicated core on InfiniBand on 8 nodes (except top right: 64 nodes)

7.4 | Results with ~~h~~Hardware-~~b~~Based pProgression

Bull BXI[23] network is designed to handle progression in hardware. Overhead ratios for Bull BXI v1.2 interconnect with OpenMPI 2.0.4.5-bull on machine irene/bxi are shown on Figure 25.

We observe mostly a sequential behavior on `MPI_Ialltoall` and other collectives. The worst case happens with `MPI_Ibcast`, where the ratios show a slowdown compared to sequential execution. This strange behavior was observed only for this collective.

BXI is designed to handle collective offload to the network card. Unfortunately BXI v1.2 cards do not properly support this feature. BXI v2 cards have been announced, and should fix support for hardware-assisted progression for collectives, so as to reach better overlap in the future.

8 | CONCLUSION

Collective communications are one of the most important features of MPI. Since version 3.0 of the standard, all collective operations have a nonblocking version. One of the goals of nonblocking communication is to enable computation/communication overlap. However, overlapping communication with computation requires a progress engine within the MPI library and its runtime system. As state-of-the-art metrics show incomplete and sometimes misleading information about the behavior of MPI runtime systems when performing nonblocking collectives, the goal of this paper is to propose a thorough methodology to study and assess the computation/communication overlap of nonblocking collectives with several new metrics. Results show that, by

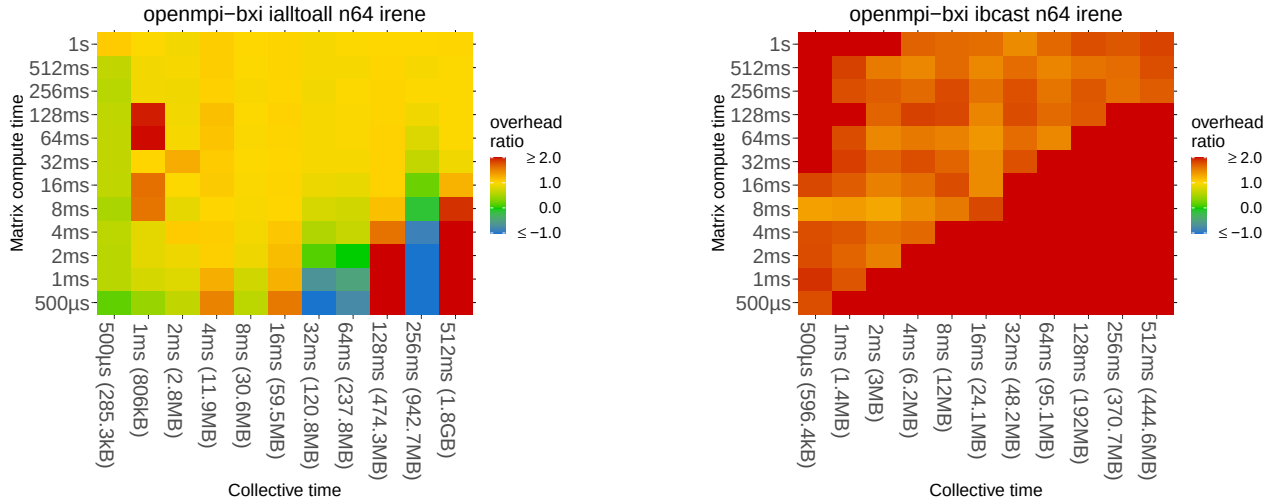

FIGURE 25 OpenMPI overlap results on BXI on 64 nodes

default, MPI implementations do not exhibit effective overlap in most cases and sometimes the use of nonblocking operations degrades the performance. The situation improves when a thread is used for progression within the MPI runtime system. In the end, acceptable performances are visible only with a progress thread bound on a dedicated core.

Such experimental finding leads to a very interesting question: if it is required to dedicate a core for nonblocking collective progression, how to design a trade-off for efficient progression and efficient computation? Maybe, with the advent of processors featuring many cores, deciding when dedicating a core for communication progression will lead to the right solution. We will investigate this option in our future work.

All results and benchmark source code have been publicly released[26].

AUTHORSHIP STATEMENT

Alexandre Denis: Conceptualization, Methodology, Resources, Writing - Original Draft.
Julien Jaeger: Conceptualization, Resources, Writing - Original Draft.
Emmanuel Jeannot: Conceptualization, Supervision, Writing - Original Draft.
Florian Reynier: Conceptualization, Methodology, Software, Investigation, Writing - Original Draft.

ACKNOWLEDGMENTS

This work was granted access to the HPC resources of TGCC under the allocation 2021- A0100601567 attributed by GENCI (Grand Equipement National de Calcul Intensif).

Some experiments presented in this paper were carried out using the PlaFRIM experimental testbed, supported by Inria, CNRS (LABRI and IMB), Université de Bordeaux, Bordeaux INP and Conseil Régional d'Aquitaine (see `https://www.plafrim.fr/`).

List of changes

References

1. MPI Forum . *MPI - a Message Passing Interface Standard V3.0. Ean 1114444410030.* English Book Service . 2012.

2. Wickramasinghe US, Lumsdaine A. A Survey of Methods for Collective Communication Optimization and Tuning. *ArXiv* 2016; abs/1611.06334.

3. Bernholdt D, Boehm S, Bosilca G, et al. A survey of MPI usage in the US exascale computing project. *Concurrency and Computation: Practice and Experience* 2018; 32. doi: 10.1002/cpe.4851

4. MPI Forum . *MPI: a Message Passing Interface Standard V1.0.* University of Tennessee, Knoxville . 1993.

5. Denis A, Trahay F. MPI Overlap: Benchmark and Analysis. In: *International Conference on Parallel Processing (ICPP)*; 2016; Philadelphia, United States.

6. Hoefler T, Lumsdaine A. Message progression in parallel computing - to thread or not to thread?. In: *2008 IEEE International Conference on Cluster Computing*; 2008: 213–222

7. Denis A. pioman: a pthread-based Multithreaded Communication Engine. In: *Euromicro International Conference on Parallel, Distributed and Network-based Processing*; 2015; Turku, Finland.

8. Denis A, Jaeger J, Jeannot E, Pérache M, Taboada H. Study on progress threads placement and dedicated cores for overlapping MPI nonblocking collectives on manycore processor. *The International Journal of High Performance Computing Applications* 2019; 33(6): 1240–1254. doi: 10.1177/1094342019860184

9. Hoefler T, Lumsdaine A, Rehm W. Implementation and performance analysis of non-blocking collective operations for MPI. In: *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*- IEEE. ; 2007: 1–10

10. Friedley A, Bronevetsky G, Hoefler T, Lumsdaine A. Hybrid MPI: efficient message passing for multi-core systems. In: *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*- IEEE. ; 2013: 1–11.

11. Si M, Peña AJ, Balaji P, Takagi M, Ishikawa Y. MT-MPI: Multithreaded MPI for Many-Core Environments. In: *Proceedings of the 28th ACM International Conference on Supercomputing*ICS '14. Association for Computing Machinery; 2014; New York, NY, USA: 125–134

12. Carribault P, Pérache M, Jourdren H. Enabling Low-Overhead Hybrid MPI/OpenMP Parallelism with MPC. In: Sato M, Hanawa T, Müller MS, Chapman BM, de Supinski BR., eds. *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More, Proceedings of the 6th International Workshop on OpenMP (IWOMP 2010)*. 6132 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg. 2010 (pp. 1–14)

13. Hjelm N, Dosanjh MGF, Grant RE, Groves T, Bridges P, Arnold D. Improving MPI Multi-Threaded RMA Communication Performance. In: *Proceedings of the 47th International Conference on Parallel Processing*- ICPP 2018. Association for Computing Machinery; 2018; New York, NY, USA

14. Ruhela A, Subramoni H, Chakraborty S, Bayatpour M, Kousha P, Panda DK. Efficient design for MPI asynchronous progress without dedicated resources. *Parallel Computing* 2019; 85: 13–26. doi: https://doi.org/10.1016/j.parco.2019.03.003

15. Intel Corporation . IMB-NBC Benchmarks. `https://github.com/intel/mpi-benchmarks`; .

16. The Ohio State University . OMB (OSU Microbenchmarks). `http://mvapich.cse.ohio-state.edu/benchmarks/`; .

17. Hoefler T, Schneider T, Lumsdaine A. Accurately Measuring Overhead, Communication Time and Progression of Blocking and Nonblocking Collective Operations at Massive Scale. *International Journal of Parallel, Emergent and Distributed Systems* 2010; 25(4): 241–258.

18. Reussner R, Sanders P, Träff JL. SKaMPI: A Comprehensive Benchmark for Public Benchmarking of MPI. *Sci. Program.* 2002; 10(1): 55–65. doi: 10.1155/2002/202839

19. Hunold S, Carpen-Amarie A, Träff JL. Reproducible MPI Micro-Benchmarking Isn't As Easy As You Think. In: *Proceedings of the 21st European MPI Users' Group Meeting*EuroMPI/ASIA '14. Association for Computing Machinery; 2014; New York, NY, USA: 69–76

20. Becker D, Rabenseifner R, Wolf F. Implications of non-constant clock drifts for the timestamps of concurrent events. In: ; 2008: 59–68

21. Hunold S, Carpen-Amarie A. On the Impact of Synchronizing Clocks and Processes on Benchmarking MPI Collectives. In: *EuroMPI*ACM; 2015: 8:1–8:10

22. Nigay A, Mosimann L, Schneider T, Hoefler T. Communication and Timing Issues with MPI Virtualization. In: *27th European MPI Users' Group Meeting*EuroMPI/USA '20. Association for Computing Machinery; 2020; New York, NY, USA: 11–20

23. Derradji S, Palfer-Sollier T, Panziera J, Poudes A, Atos FW. The BXI Interconnect Architecture. In: *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*; 2015: 18–25.

24. Mills DL. Internet time synchronization: the network time protocol. *IEEE Transactions on Communications* 1991; 39(10): 1482–1493.

25. Hunold S, Carpen-Amarie A. Hierarchical Clock Synchronization in MPI. In: *2018 IEEE International Conference on Cluster Computing (CLUSTER)*; 2018: 325–336

26. Bench NBC website. `http://pm2.gitlabpages.inria.fr/bench_nbc/`; .