Study on progress threads placement and dedicated cores for overlapping MPI nonblocking collectives on manycore processor

Alexandre Denis, Julien Jaeger, Emmanuel Jeannot, Marc Pérache, Hugo Taboada

Journal Title
XX(X):1–14

DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

Study on progress threads placement and dedicated cores for overlapping MPI nonblocking collectives on manycore processor

Alexandre Denis[2], Julien Jaeger[1], Emmanuel Jeannot[2], Marc Pérache[1], Hugo Taboada[1]

Abstract

To amortize the cost of MPI collective operations, nonblocking collectives have been proposed so as to allow communications to be overlapped with computation. Unfortunately, collective communications are more CPU-hungry than point-to-point communications and running them in a communication thread on a dedicated CPU core makes them slow. On the other hand, running collective communications on the application cores leads to no overlap.

In this paper, we propose placement algorithms for progress threads that do not degrade performance when running on cores dedicated to communications to get communication/computation overlap. We first show that even simple collective operations, such as those based on a chain topology, are not straightforward to make progress in background on a dedicated core.

Then, we propose an algorithm for tree-based collective operations that splits the tree between communication cores and application cores. To get the best of both worlds, the algorithm runs the short but heavy part of the tree on application cores, and the long but narrow part of the tree on one or several communication cores, so as to get a trade-off between overlap and absolute performance. We provide a model to study and predict its behavior and to tune its parameters.

We implemented both algorithms in the MPC framework, which is a thread-based MPI implementation. We have run benchmarks on manycore processors such as the KNL and Skylake and get good results for both performance and overlap.

Keywords

Non-blocking Collectives, MPI, Placement, Communication/Computation Overlap

1 Introduction

MPI[13] is the standard interface for communications in HPC applications. It is used by applications for inter-node (i.e. network) and intra-node (processes on the same node) communications. The cost of communications is one of the main obstacles to get a good speedup for parallel applications. To amortize the cost of MPI communications, application programmers try to overlap communications with computation by using nonblocking communication primitives, and let them progress in background while keeping the CPU busy with computation.

Initially the nonblocking communications were only available for point-to-point communications. The extension of the nonblocking communications to collective operations (i.e. primitives that involve more than two nodes, such as broadcast, reduce, scatter, gather, ...) is an addition of the latest major MPI version[13]. It opens the door to communication/computation overlap for collective operations too. However, collective communications are more CPU-hungry than point-to-point communications because they involve multiple point-to-point communications, and thus a lot of communication tasks. Furthermore, collective communication cannot be done with a DMA transfer programmed at the beginning once for all and let it progress in background; their algorithm may need to receive and send data at any point and thus need CPU intervention at these points. Therefore it is harder to make collective communication progress in background. Since most implementations of the nonblocking MPI collectives do not progress efficiently in background, very few applications actually use them; in these few codes[6], the time spent in the communication part is insignificant compared to the computation time, which doesn't encourage MPI implementer to improve progression in the nonblocking collectives. Our contribution aims at being a first step to break this chicken-and-egg situation.

In this paper, we tackle the problem of overlapping communication and computation for nonblocking collectives on manycore processors. Since we consider only intra-node communications, all communications are memory copies. Therefore, these communications use the CPU resources and cannot be done at the same time as computation. We study the case of MPI tasks spread on a manycore processor, with one task per core, and how to improve overlap with cores dedicated to communications. We show

[1]CEA, DAM, DIF, F-91297 Arpajon, France
[2]Inria, LaBRI, Univ. Bordeaux, CNRS, Bordeaux-INP, France

Corresponding author:
Hugo Taboada, CEA, DAM, DIF, F-91297 Arpajon, France
Email: hugo.taboada@cea.fr

that a naive approach to have communication progress in background using only dedicated communication cores makes communication actually slower. We explore specific placement of progress threads depending on the topology of the algorithm used by the collective operation. We consider two classes of algorithms: chain-based algorithms, for which we show that contrary to its apparent simplicity, making it actually progress efficiently in background is not straightforward; and tree-based algorithms, which impose variable load in time on communication cores, that we will exploit to reach a trade-off between fast communication and good overlap.

In short, this paper makes the following contributions:

- we propose a *placement* scheme for progress threads in chain-based collective communications;
- we propose an *algorithm* that splits the tree of tree-based collective operations, running parts of the tree on cores dedicated to communication, and parts of the tree on the application core;
- we propose a *model* for the above algorithm, so as to demonstrate the improvement of global performance when overlapping communication and computations, and to tune its parameters independently from machine performance;
- we *implemented* the algorithm in the MPC[14] MPI implementation and compared it with other existing MPI implementations.

The rest of the paper is organized as follows. Section 2 presents related work about communication/computation overlap in general, and for collective communication in particular. Section 3 presents the MPC framework which is the context of our work and hosts the implementation of algorithms described in this paper. Section 4 describes our initial progress threads placement. Section 5 study the odd-even placement strategy for chain-based collective operations. Section 6 presents our split-tree algorithm for tree-based collective communications, a model of the algorithm and how to tune it for optimal performance. Section 7 reports experimental results, and Section 8 concludes.

2 Related Work

In this Section, we present other work related to MPI communication progression.

The topic of communication progression has already been studied for some aspects in the literature. Several strategies do exist for background progression of point-to-point communications, such as offloading the communication to hardware[15,19] and let the hardware do the progression; use of a thread[7,16] or process[11] dedicated to communication progression; opportunistic scheduling of communication tasks[4,18].

MPI nonblocking collective communications are more difficult to make progress in the background, since not only the data transfer but the collective algorithm too needs to progress, which makes it harder to rely on hardware. There is specific work[1] for hardware-assisted progression on IBM Blue Gene and on Bull BXI[5], or offloading shared memory collectives to a kernel module[12] (although authors only address performance of blocking collectives, not progression of nonblocking collectives). The reference NBC implementation[9] relies on a progression thread, with some tricks[8] to improve overlap on InfiniBand. This approach is quite different from ours since it leads to one progression thread per MPI task, while our approach runs multiple MPI ranks in the same process and the algorithm for the collectives is shared across all MPI ranks in the same process, which allows for a coordinated strategy between MPI ranks.

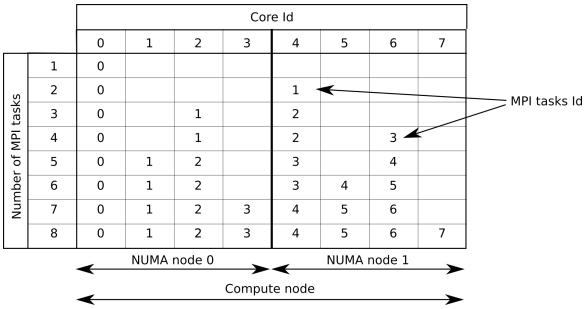


Figure 1. MPI tasks placement policy in MPC on a 8-core node with two NUMA nodes.

3 Context of study: the MPC framework

In this Section, we present MPC[14], our thread based MPI implementation.

In MPC, MPI tasks are implemented with threads. MPC also implements POSIX threads and an OpenMP runtime system. MPC has its own user-space thread scheduler allowing a fine-grained scheduling of all these threads. Thus, we bypass the system scheduler. MPC uses a tuned version of libNBC[9] to implement MPI 3 Non-Blocking Collectives. One progress thread is created for each MPI task. In this implementation, a MPI nonblocking collective is decomposed in MPI point-to-point nonblocking calls fulfilling the collective algorithm. When a MPI nonblocking collective is called, each MPI task creates a *schedule* containing requests for the point-to point nonblocking calls corresponding to its part of the collective algorithm, and attach it to its associated progress thread. Thus, the progress threads handle the communication described by the schedules while MPI tasks continue to execute computation.

4 Application threads placement and communication cores.

In this paper, we focus on intra-node communications on a manycore machine, with one MPI task per core. This addresses the case of pure single-node deployment but is beneficial for inter-node too since collective algorithms running on clusters are built upon intra-node operations for their local part on each node.

The default placement in MPC for these MPI tasks is to evenly spread unused cores in a NUMA node. This way, each MPI task has (nearly) the same resources to spawn threads. Figure 1 shows a summary of MPI tasks placement policy in MPC, on a 8-core node with two NUMA nodes. With this placement, if one has only 4 MPI tasks on the node, MPI tasks will be spaced with one free core in-between. Contrary

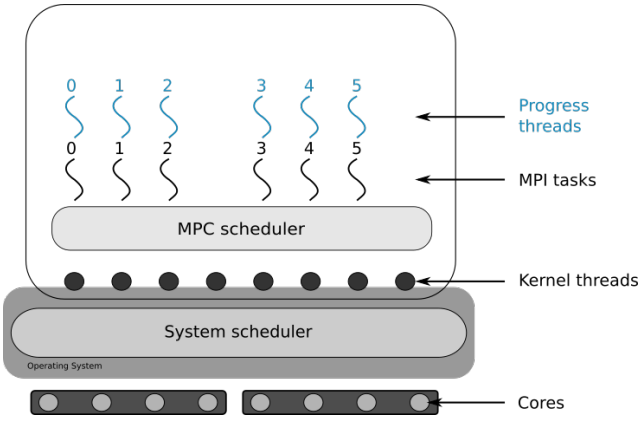


Figure 2. *bind* placement policy of progress threads

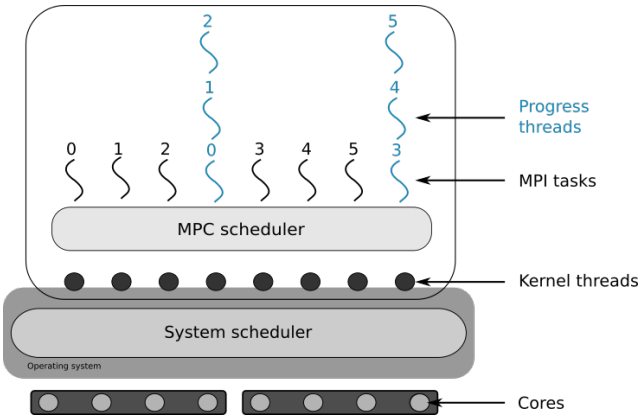


Figure 3. *numa* placement policy of progress threads

to a *scatter* policy, MPI tasks with ranks 0 and 1 will remain on the same NUMA node. In another example with 6 MPI tasks, each NUMA node still have one free core.

As explained in the previous Section, each MPI task spawns a progress thread to handle nonblocking collective communications. These progress threads can be bound through different algorithms. In the default behavior used in our experiments, MPI tasks are bound with the previous policy and progress threads are bound to the same core as their parent MPI tasks. This behavior can be found in several MPI implementations, and is shown in Figure 2 for 6 MPI tasks on a 8-core node. We call this behavior the *bind* placement policy.

Communication cores. To obtain a good overlap for communications and computation, the progress threads have to run at the same time as application threads. On a manycore machine, the straightforward way to get background progression of communication is to dedicate some cores to communications, thus some cores host an MPI rank, we call them *application cores*; the remaining cores (one or several) host communication progression threads, we call them *communication cores*.

With our MPI tasks placement, cores are left unused next to the MPI tasks. These free cores are perfect candidate to host the progress threads. Hence, we propose a new placement where MPI tasks are bound with the previous policy and progress threads are bound to the closest idle cores of the same NUMA node. With this placement policy, several progress threads may be hosted on the same core, as shown in Figure 3 for 6 MPI tasks on a 8-core node. Since only one core is free for each NUMA node, all the progress threads spawned by the MPI tasks on this NUMA node are gathered on the available core. If no core is available, progress threads are bound on the same core as their MPI task.

This placement, called the *numa* placement policy, allows communication threads to be scheduled independently from application threads, so as to get overlap between communications and computation. It is noted P_{numa} and is given in equation 1 below.

$$P_{\text{numa}}(N,n,M) = \begin{cases} M & \text{if: } n \geq N \\ \left\lceil \left(\left\lfloor \frac{M}{\delta} \right\rfloor + 1\right) \times \delta \right\rceil - 1 & \text{otherwise} \end{cases} \quad (1)$$

with $\delta = \frac{N}{N-n}$ and:

N the number of cores per NUMA node;

n the number of MPI tasks on the local NUMA node;

M the position of the spawning MPI task in the NUMA node.

Two problems arise when gathering several progress threads on fewer cores.

First, operations of the progress threads on the same core are serialized. Hence, the same amount of operations to perform the collective communications may take a longer time to execute. To circumvent that effect, it may be necessary to improve the naive "numa" placement to reorder the progress threads on the communication cores according to the communication pattern of the collective and which operations are supposed to run simultaneously. This first issue and the proposed improved placement are studied in Section 5 for a chain-based collective communication pattern.

The second issue is due to the folding of the collective algorithm on a few cores. If at a given step, the algorithm of the collective involves more point-to-point communications than the number of communication cores, once folded the communication will be slower since not all communications of the step will be performed at the same time. To reduce this impact, we propose a method to select the amount of communications to be done on the communication cores for tree-based communication pattern, according to the number of communication cores. This second issue and the proposed algorithm are described in Section 6.

5 Odd-even algorithm for chain-based MPI collective operations

In this section, we present an algorithm to mitigate the performance impact of executing a chain-based MPI collective operation only on communication cores.

5.1 Study of chain-based collective operations

The collective operation `MPI_Iscan` (parallel prefix) may be implemented with various algorithms[17]: chain, binomial tree, simultaneous binomial trees. We will use the chain topology since our nonblocking collective implementation

derives from libNBC[9] which implements it as a chain. For a binomial tree, it would follow our split-tree algorithm described in Section 6. Simultaneous binomial trees is not considered since we work on intra-node communication and such algorithm would cause memory contention with too many communications at the same time.

The overall communication time measured for the "numa" placement, on a 64-core KNL, with 62 MPI tasks (thus 2 communication cores for progress thread) is 3 times slower than our original "bind" placement. This result can be seen on the first two bars of Figure 7 on page 6.

To understand why the "numa" placement leads to slow communication compared to the "bind" placement, we take traces of the thread scheduling. To do so, we insert trace points to track which thread is scheduled between context switches of the MPC scheduler, using the Pajé format[3]. Then we use the software ViTE[2] to view traces in this format. This allows us to see which thread is running on which core at any time. In these traces, colors correspond to the types of threads running on cores:

- ■ represents MPI tasks running computation.
- ■ represents progress threads running MPI nonblocking collectives.
- ■ represents busy waiting of internal MPC threads. In this case, we see a `MPI_Barrier` just after the `MPI_Wait` call.
- ■ represents the "idle thread". This is the thread that is executed when the MPC scheduler has no thread to run.

Figure 4 depicts the execution trace of chain-based `MPI_Iscan` algorithm with the "bind" progress thread placement, for 63 MPI tasks on a 64-core KNL. Each row represents the state of a core; progress threads are bound to the same core as the MPI task that spawned it.

In the chain-based algorithm, each progress thread associated with MPI tasks 1 to $n-2$ receives a message, and then, sends a message to the next MPI rank. The progress thread associated with MPI task 0 only sends one message while the progress thread associated with MPI task $n-1$ performs only one message reception. The blue diagonal that we observe in the trace corresponds to these message exchanges. In the zoom box, we can see this communication pattern for each thread, with two blue boxes corresponding to the 2 phases of the communication scheme: first receive, then send. Because the progress thread shares the same core as the MPI task, it is difficult to overlap communication and computation on the core since both threads cannot be scheduled at the same time. However, we notice that blue boxes from different MPI tasks are executing simultaneously. It means that communication phases – send and receive – from different MPI tasks are executed at the same time.

In order to have a better overlap between communications and computation, we execute the `MPI_Iscan` algorithm with the "numa" placement of progress threads. We can see the execution trace of `MPI_Iscan` running 62 MPI tasks on a 64-core KNL with the "numa" placement in the Figure 5. Since free cores are evenly spread and we bind each progress thread to the nearest communication core, progress threads 0 to 30 are bound on core 31 and progress threads 31 to 61 are bound on core 63.

With this placement for chain-based algorithm, cores 31 and 63 are used only half of the time (in blue) and are idle the rest of the time (in orange). The chain-based pattern enforces that the first rank to communicate is 0, then 1, then 2, up to 30. For all these communications, the progress threads in charge of it are all on core 31 and all the progress threads bound to core 63 are idle at this point in the algorithm.

The problem with this placement is that consecutive ranks are bound to the same core, thus cannot be active at the same time, one for send, the other for receive. We observed with the "bind" placement that communication phases from consecutive ranks can execute concurrently. However, with most of these threads being bound to the same core, every communication phase is serialized. Each non-concurrent communication phase is aggregating, thus increasing the overall time of the whole communication. Furthermore, we also observe that only one concurrent execution of communication phases is still happening, between MPI task 30 and MPI task 31. When the progress thread associated with MPI task 30, running on the core 31, sends a message, the progress thread associated with MPI task 31 is already running on the core 63 to receive the message. Some parts of the algorithm can run simultaneously. This observation shows us that one core is not sufficient to perform chain-based communication algorithms since it does not allow to run these parts simultaneously. Indeed, if we only had one core for progress threads, 2 progress threads could not run at the same time. Also, even with more communication cores, consecutive MPI ranks should not be bound to the same core, to avoid this exact behavior.

5.2 Odd-even placement for chain-based collectives

To tackle this problem, we propose the "odd-even" algorithm shown in Equation 2. The algorithm takes as input the number of cores p, the number of MPI tasks n and the spawning MPI rank r to compute the progress thread placement.

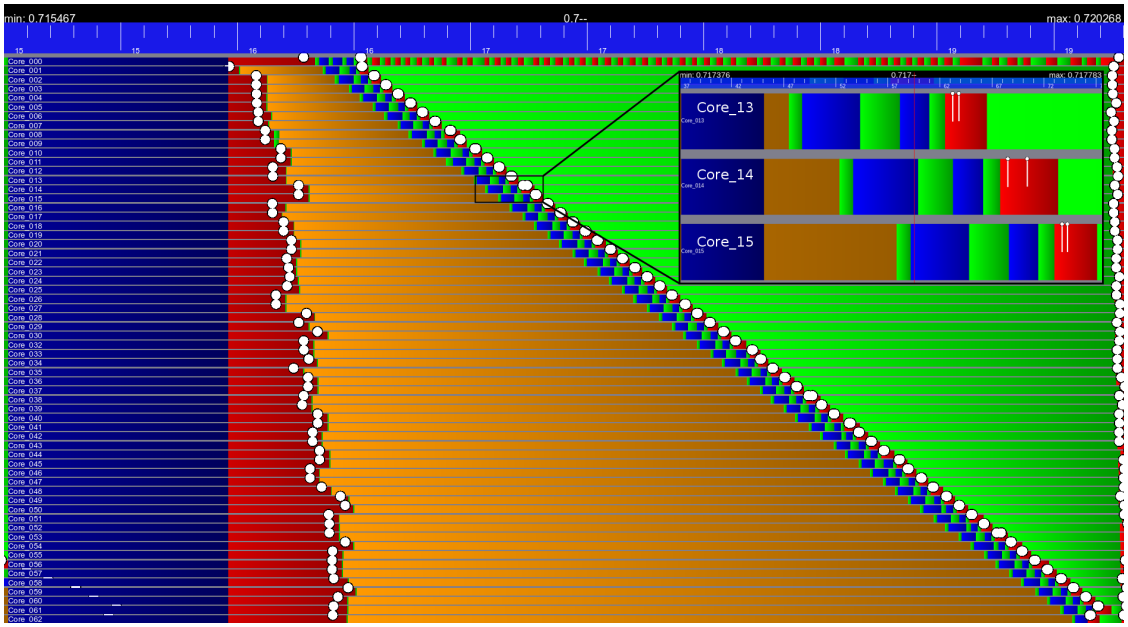
$$P_{\text{odd-even}}(p,n,r) = \frac{p}{p-n} \times (r \bmod (p-n)) + \frac{p}{p-n} - 1 \tag{2}$$

In our study, two cores are dedicated for progress threads. Then, instead of performing the naive "numa" placement, we reorganize the progress threads on these two communication cores so that a receive gets posted on one core and the corresponding send is always posted on the other core. Consecutive MPI ranks should not be on the same core, thus it means rank 0 should not be on the same core as rank 1, and the same for rank 1 and rank 2. However, there is no restriction for rank 0 and rank 2.

Thus, we realized an odd-even distribution of progress threads. All progress threads associated with odd MPI ranks are bound on one communication core and the progress threads associated with even MPI ranks are bound on the other communication core.

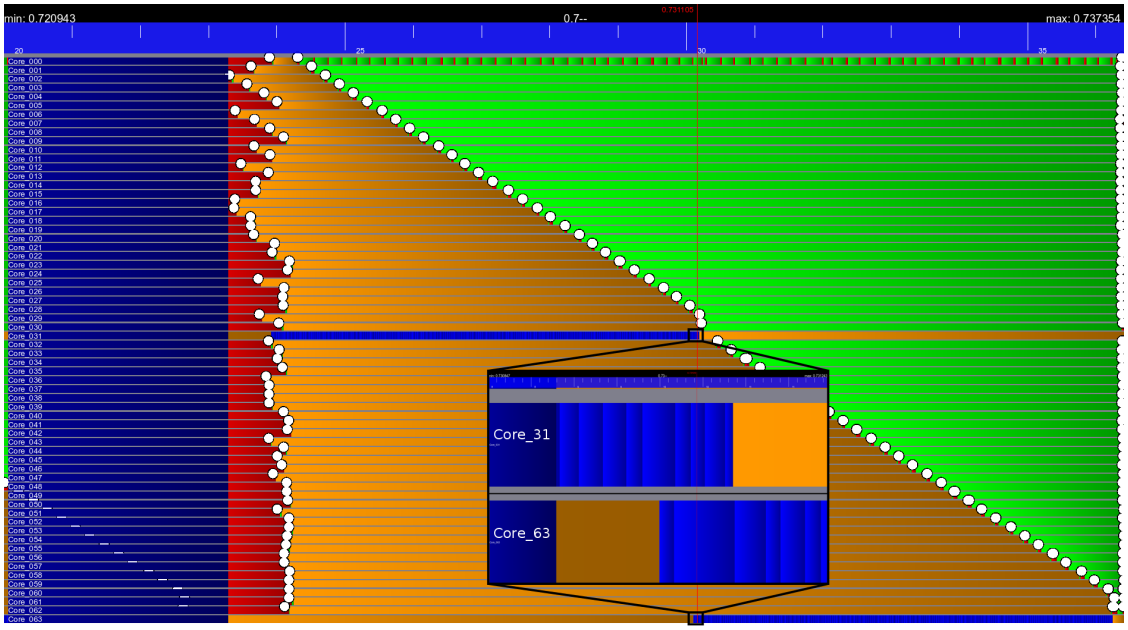
5.3 Experimental results

The execution trace with our "odd-even" placement is depicted in Figure 6. We can see that unlike the "numa"



Color caption:
MPI tasks: Computation
Progress threads: Communication
Internal MPC thread: Busy waiting
Thread idle: Passive waiting

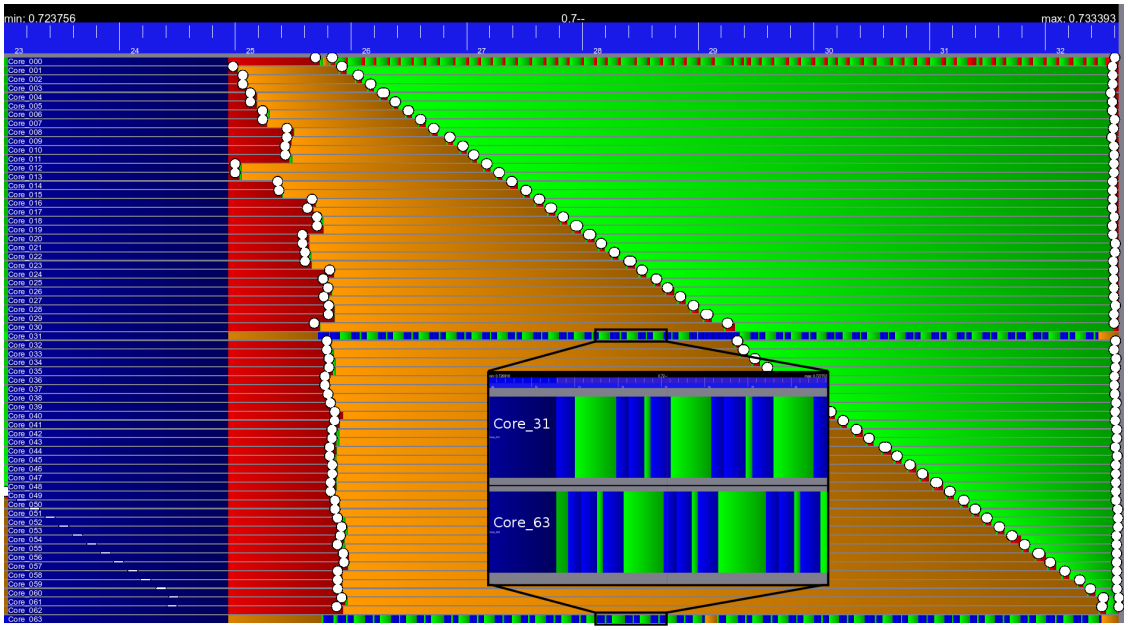
Figure 4. Execution trace of `MPI_Iscan` with the "bind" progress threads placement with 63 MPI tasks.



Color caption:
MPI tasks: Computation
Progress threads: Communication
Internal MPC thread: Busy waiting
Thread idle: Passive waiting

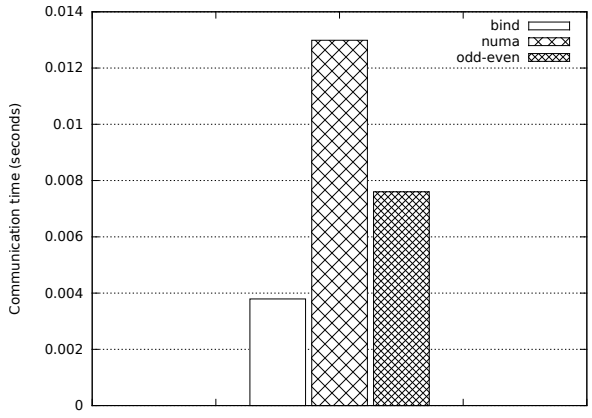
Figure 5. Execution trace of `MPI_Iscan` with the "numa" progress threads placement on 64 cores (62 MPI tasks, 2 communication cores).

placement which allows to run concurrently only a small number of communication phases (between MPI rank 30 and MPI rank 31), the placement "odd-even" allows concurrency for all message exchanges in the collective. The send phases of even-ranked MPI tasks (respectively odd-ranked) overlap with the reception phases of odd-ranked MPI tasks (respectively even-ranked).



Color caption:
MPI tasks: Computation
Progress threads: Communication
Internal MPC thread: Busy waiting
Thread idle: Passive waiting

Figure 6. Execution trace of `MPI_Iscan` with the "odd-even" progress threads placement on 64 cores (62 MPI tasks, 2 communication cores).



Overlap:

- bind: communication time *not overlappable*
- numa: communication time *overlappable*
- odd-even: communication time *overlappable*

Figure 7. Communication time of `MPI_Iscan` for 62 MPI tasks on 64 cores with a buffer of 1MB.

We measure the communication time with these three different placements: "bind", "numa" and "odd-even". Communication time with these placements for 62 MPI tasks on a 64-core KNL are depicted in Figure 7. The "odd-even" placement is twice as fast as the "numa" placement but is still less efficient than "bind". However, contrary to the "bind" placement, these communications are still overlappable, and less computation is required to completely hide them than with "numa" placement.

The global behavior of the "odd-even" algorithm is a better overlap than both "bind" and "numa" algorithms and thus gets an overall better performance when communication is performed at the same time as computation, even though communication takes a longer time when executed alone. For non-blocking communication with actual computation done between communication start and wait, the "odd-even" algorithm will always get a better overall performance than both "bind" and "numa" placements.

We have shown that even the simple chain-based collective operations are not straightforward to schedule on dedicated communication core so as to get communication/computation overlap, since a naive placement get low performance. We have proposed a placement strategy that allows the send and the receive part of each point-to-point communication to be scheduled at the same time, which greatly improves communication performance. Since this placement uses only dedicated communication cores, it able to get good overlapping.

6 A split-tree algorithm for tree-based MPI collective operations

In this Section, we propose a novel algorithm for tree-based MPI collective communications which improves communication/computation overlap.

6.1 The split-tree algorithm

Tree-based collective communication algorithms involve a larger amount of point-to-point communications, and thus a lot of communication tasks. When communication cores

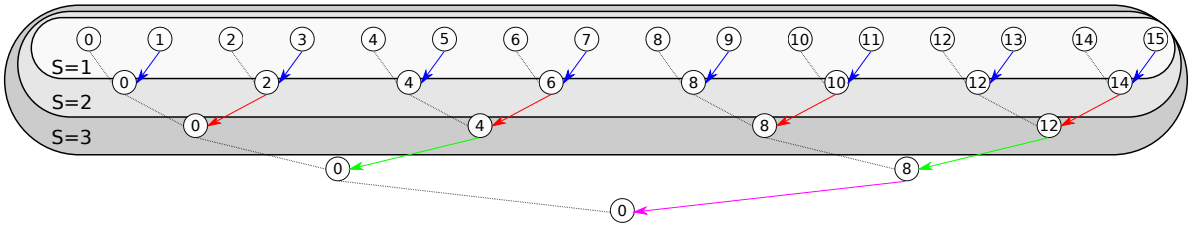


Figure 8. Communication tree for a reduce collective with 16 MPI tasks. S is the number of steps (tree levels) running on application cores. Plain edges are communications. Vertices are the MPI tasks; numbers in vertices are MPI ranks

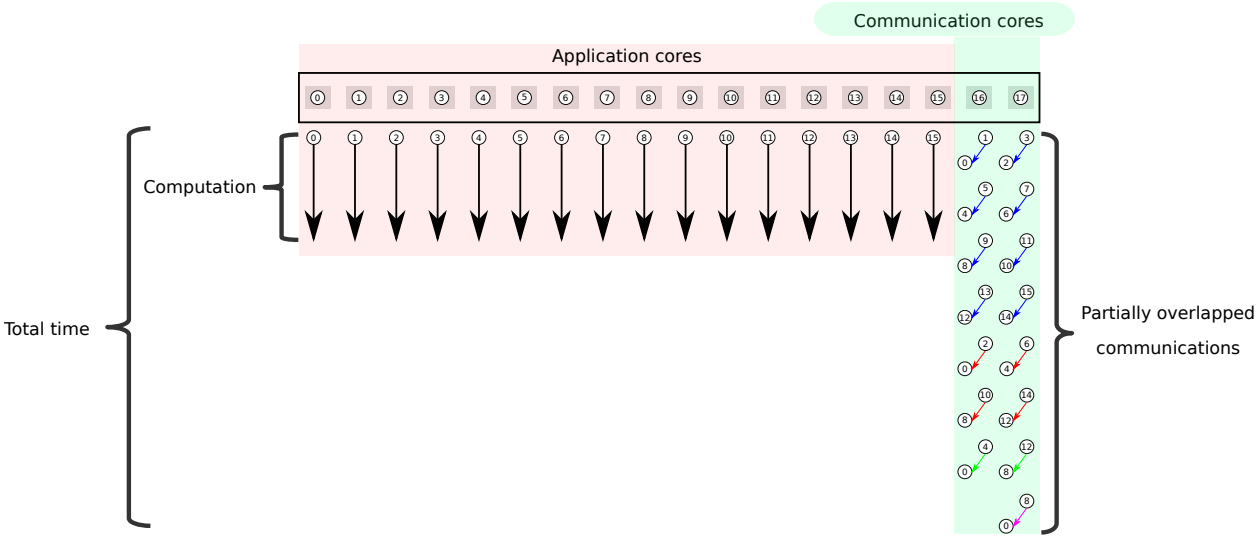


Figure 9. Example of all communication on 2 communication cores on a machine with 18 cores.

perform all communications on behalf of all application cores, the algorithm is *folded* and communications from a given step of the collective algorithm may be serialized. As a consequence, when folded on few communication cores, collective communications get much slower than when executed as a blocking call on all application cores simultaneously.

In this Section, we consider *tree-based* algorithms (reduce, broadcast, gather, scatter, allreduce). The time steps of such a tree-based collective are depicted in Figure 8: each level of the tree is a step in the algorithm, from the leaves to the root for the given example of a *reduce* operation. The rank of MPI tasks participating to each step is represented in the vertices. The left child of a vertex is the same MPI task; only the right child involves a communication. When represented as time steps of the algorithm, it is a binary tree, although when considering the data flow by duplicating vertices which are the same task, the algorithm is really a binomial tree.

If we execute this collective algorithm folded on the communications cores, we get operations scheduled as depicted in Figure 9: computation can run on application cores, and be fully overlapped with communications. However, when folded on a few cores, the collective communication will run much slower because each round cannot be run at once and operations get serialized. On the example tree in Figure 8, step #1 in blue is made of 8 point-to-point communications; when folded on 2 communication cores in Figure 9, this step alone consumes 4 rounds, and the overall performance of the collective is poor: it takes 8 rounds instead of 4.

On such tree-based algorithms, we observe that the amount of work is very unbalanced in time and space. On our example for 16 MPI tasks, there are 15 communication tasks and the algorithm needs 4 steps. If we fold these communications on a single communication core, it would need 15 steps which is 4 times slower. Since half of the work is in the first step, represented as $S = 1$ with levels numbered from the leaves, we can trade some performance against some overlap by executing different parts of the tree on different cores. If only the lower part of the tree is executed on the communication cores, and the first step $S = 1$ is executed on the application cores, then the total is twice as fast as running everything on communication cores, while only a single step cannot be overlapped with computation. For operations up to $S = 2$ executed on application cores and the remaining operations are executed on communication cores, we get the example shown in Figure 10; on this example, although not all communications are overlappable, the total time is better than what we obtained with all communications running on communication cores as in Figure 9.

Our proposed algorithm is a generalization of this principle for a *trade-off between communication performance and overlap*: split the communication tree with the lower part running on communication cores, so as to have full overlap, and the upper part running on all application cores, to benefit from parallelism. Let S the number of steps (tree levels) running on application cores. The algorithm runs S steps of the tree on application cores with level numbers as depicted in Figure 8. The case of $S = 0$ is equivalent to

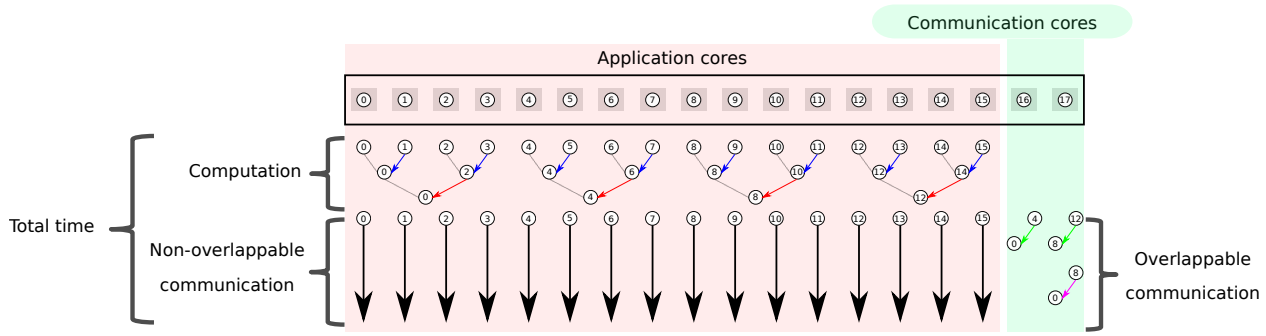


Figure 10. Example for split-tree algorithm with $S = 2$ and $N = 16$ on a machine with 18 cores.

running all the communication on communication cores as already seen in Figure 9. When $S = 1$, the algorithm runs the short but heavy part of the tree on application cores whereas the long but narrow part of the tree is running on one or several communication cores. All the communications running on application cores cannot be overlapped by computation because they are running on the same cores and they are memory copies. However, this part of the tree is the heaviest and running these communications on few communication cores would jeopardize communication performance. The part of the tree running on communication cores benefits of total overlapping of its communications.

If S is increased, the algorithm loses a bit of its ability of being overlapped but can increase its absolute performance depending on the communication/computation ratio. We have to get a trade-off between overlap and absolute performance.

6.2 Algorithm Modeling and Tuning

In this Section, we propose a performance model of the algorithm described in Section 6.1, so as to show its relevance and to tune its S parameter.

Model for collective operations. Let N_{proc} the total number of cores, and N the number of cores for the application (i.e. number of MPI ranks), then the number of dedicated cores for communication is $P(N) = N_{proc} - N$.

We consider collective operations as binomial trees only. The proposed model could be easily extended to m-nomial trees if needed, for arbitrary m. It applies to operations such as: reduce, broadcast, gather, scatter. We model communication cost as linear, neglecting latency and cache effects. We take as unit the point-to-point transfer time of one buffer of the size of the considered collective operation. We study first operations with a constant buffer size across the whole tree (reduce, broadcast). We will extend it to variable-buffer size operations (scatter, gather) in a second step.

The height of the tree* is $H(N) = \lceil \log_2(N) \rceil$. In the case of a blocking operation where communication is performed simultaneously by all application cores, we get the following execution time:

$$T_{blocking}(N) = H(N) = \lceil \log_2(N) \rceil \tag{3}$$

Let $C(N)$ the computation time on N nodes. To model computation and communication overlap, we consider the application programmer tried to reach perfect overlap and sized computation to have the same duration on all cores as the blocking collective operation, i.e. $C(N_{proc}) = T_{blocking}(N_{proc})$. If we assume computation scales linearly, we have the following time for computation on N nodes:

$$C(N) = \frac{N}{N_{proc}} \times C(N_{proc}) \tag{4}$$

Model for the proposed algorithm. We now model the split tree algorithm itself. As defined in Section 6.1, S is the number of steps running on application cores; the time to run these steps is the depth of the sub-trees, namely S, unless the tree height is smaller than S. The algorithm schedules operations from the lower $H(N) - S$ levels on communication cores, folded on $P(N)$ cores.

Let $F(N, i)$ the number of communications for N MPI tasks in the level i. If N is a power of 2, the number of edges in level i is $\frac{N}{2^i}$: the first level has $\frac{N}{2}$ edges, and the number of edges is halved at each level. In the general case with N not beeing a power of 2, there are additional edges needed to connect all leaves, as shown in dashed blue lines on the example of a tree of size 23 in Figure 11. Taking into account these additional edges, actually cases where a level contains an odd number of vertices, we get as the number of communications for N MPI tasks in the level i:

$$F(N, i) = \left\lfloor \frac{N}{2^i} + \frac{1}{2} \right\rfloor \tag{5}$$

with $1 \leq i \leq H(N)$

The value of $F(N, i)$ is shown in Figure 12 for N between 1 and 64 and i between 1 and $H(64) = 6$. As expected, we observe that at each step (increasing i), the number of communications is halved.

Since each level of the tree contains $F(N, i)$ communications for level i numbered from 1 to $H(N)$, it takes a time of $\lceil F(N, i)/P(N) \rceil$ once folded on $P(N)$ communication cores, assuming each level is run in sequence because of communication dependencies. As a result, the time for a non-blocking collective with split-tree algorithm is Equation 6 as below:

*We use a binomial tree where the N MPI tasks are leaves. In case of a binary tree, we will have N vertices and $H(N) = \lceil \log_2(N + 1) \rceil - 1$

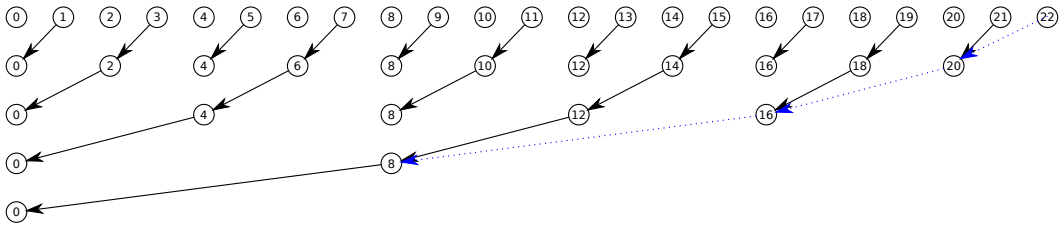


Figure 11. Communication on each level for a binomial tree with 23 MPI tasks.

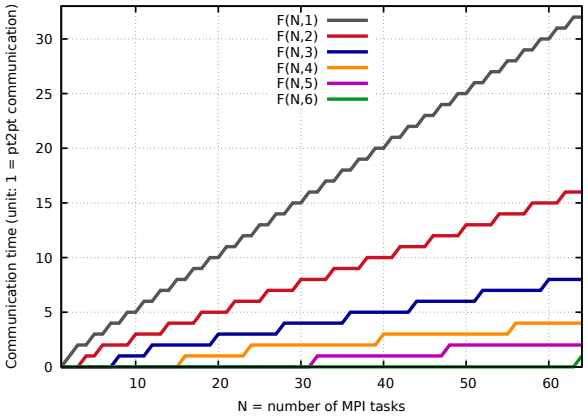


Figure 12. Number of communications for each step in the communication tree, for N from 1 to 64.

$$T_{non-blocking}(S,N) = \underbrace{\min(S, H(N))}_{\text{first } S \text{ steps}} + \underbrace{\sum_{i=S+1}^{H(N)} \left\lceil \frac{F(N,i)}{P(N)} \right\rceil}_{\text{last } H(N)-S \text{ remaining steps}} \quad (6)$$

With communication and computation overlap with the same collective operation, given that the part running on application cores cannot be overlapped and the part running on communication cores is fully overlapped, we get the result in Equation 7 as time for overlapped computation and nonblocking collective with split tree:

$$T_{overlapped}(S,N) = \underbrace{\min(S, H(N))}_{\text{non-overlappable comms}} + \underbrace{\max\left(C(N), \sum_{i=S+1}^{H(N)} \left\lceil \frac{F(N,i)}{P(N)} \right\rceil\right)}_{\text{overlappable communications}} \quad (7)$$

The graph of $C(N)$, $T_{blocking}$, and $T_{nonblocking}(N,S)$ for increasing values for S and $N_{proc} = 64$ is depicted in Figure 13. We observe that for large values of N (i.e. small number of communication cores), the communication cost is huge for $S = 0$ (all communication on communication cores). The cost decreases when S increases.

Figure 14 represents the total time of computation overlapped with communications when using blocking communications (computation and communication run in sequence) and when using nonblocking communications with split tree algorithm. We observe that increasing values for S increases the cost for small values of N (reduces overlap), but this cost is amortized for large values of N where the total time is dominated by the cost of the communication folded on few communication cores.

Extension to scatter and gather. We can extend the proposed model for collective operations where not all tree edges have the same weight, such as *scatter* and *gather*; when going from leaves to root, data size doubles at each level of the tree. If we modify the model for such operations, we get the following. Let $W_H(N) = \sum_{i=1}^{H(N)} 2^{(i-1)}$ the weight of communications with data size which doubles at each level of the tree. Let $W(N,S) = \sum_{i=1}^{S} 2^{(i-1)}$ the weight of steps between step 1 to S with data size which doubles at each level of the tree.

The time for a nonblocking collective with variable data size with split-tree algorithm is Equation 8 as below:

$$T^{var}_{non-blocking}(S,N) = \underbrace{min(W(N,S), W_H(N))}_{\text{first } S \text{ steps}} + \underbrace{\sum_{i=S+1}^{H(N)} \left\lceil \frac{F(N,i)}{P(N)} \right\rceil \times 2^{(i-1)}}_{\text{last } H(N)-S \text{ remaining steps}} \quad (8)$$

With communication and computation overlap with the same collective operation, given that the part running on application cores cannot be overlapped and the part running on communication cores is fully overlapped, we get the result in Equation 9 as time for overlapped computation and nonblocking collective with variable data size with split tree:

$$T^{var}_{overlapped}(S,N) = \underbrace{\min(W(N,S), W_H(N))}_{\text{non-overlappable comms}} + \underbrace{\max\left(C(N), \sum_{i=S+1}^{H(N)} \left\lceil \frac{F(N,i)}{P(N)} \right\rceil \times 2^{(i-1)}\right)}_{\text{overlappable communications}} \quad (9)$$

The graphs for scatter and gather are shown in Figure 15 for communication cost and in Figure 16 for overlapped time. The observed behavior is similar to the reduce/broadcast seen previously.

Discussion and tuning. From observation of Figure 14, the absolute minimum time is reached for $S = 0$ and $N = 51$. However, it means that 13 cores are dedicated

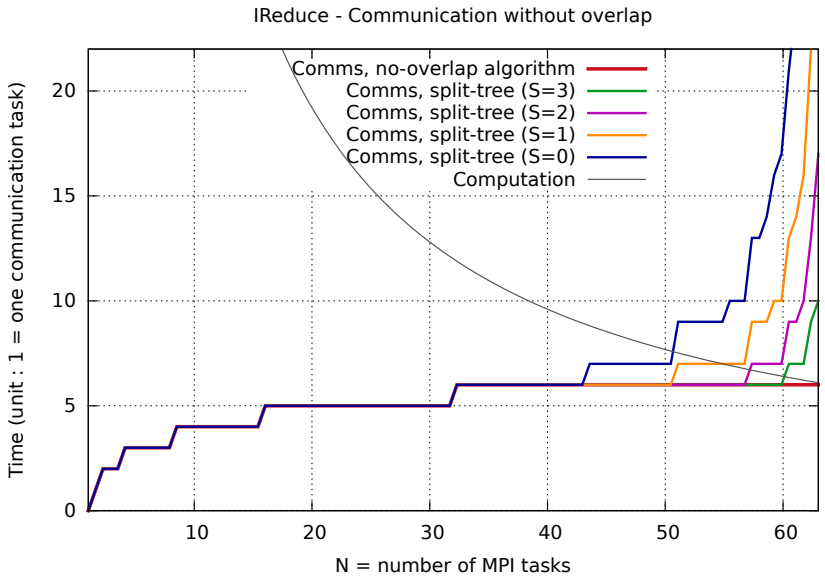


Figure 13. Model of communication cost for operations with constant-size buffer (broadcast, reduce) on 64 cores.

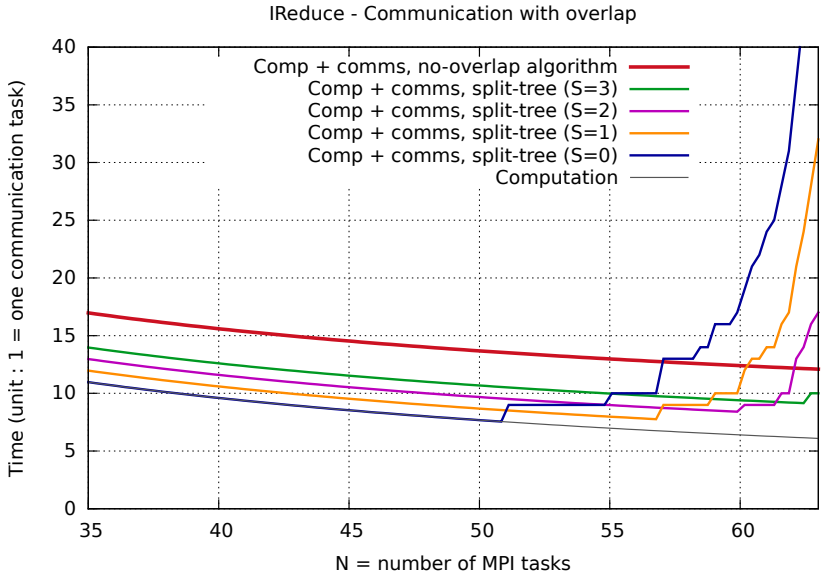


Figure 14. Model of communication/computation overlap for operations constant-size buffer (broadcast, reduce) on 64 cores.

to communications, which may not be desirable for the user since it would degrade performance of parts of the application without communication. With 7 cores dedicated to communications ($N = 57$), the optimal is $S = 1$; for 4 cores dedicated to communication ($N = 60$), the optimal is $S = 2$; and finally $S = 3$ for $N = 62$ (2 communication cores).

As a general case, for a given value of N, it is enough to compute the predicted performance with the model for a few values of S to find the optimal value. However finding N for the best overall performance depends on application scalability and communication/computation ratio and is out of scope for this paper.

6.3 Implementation in MPC

To implement our algorithms, we define the parameter S to be the number of steps (tree levels) that we want to run on application cores. For *all-to-one* algorithms (reduce, gather), we run the S steps on MPI tasks using MPI point-to-point blocking communication before creating the NBC schedule of $H(N) - S$ steps. Then, we attach it to its associated progress thread. Thus, the first part of the algorithm is running on application cores whereas the last part is running on the cores dedicated to the progress threads. For *one-to-all* algorithms (broadcast, scatter), we define the requests of $H(N) - S$ steps and create the NBC schedule first. We attach it in its associated progress thread. Then we execute the last S steps in the `MPI_Wait` function executed by the MPI tasks. Hence, the first part is running on the cores dedicated to the progress threads whereas the last part is running on application cores.

These mechanisms add a fixed overhead that is amortized for large messages. Only large messages are considered for communication/computation overlap so that the overhead

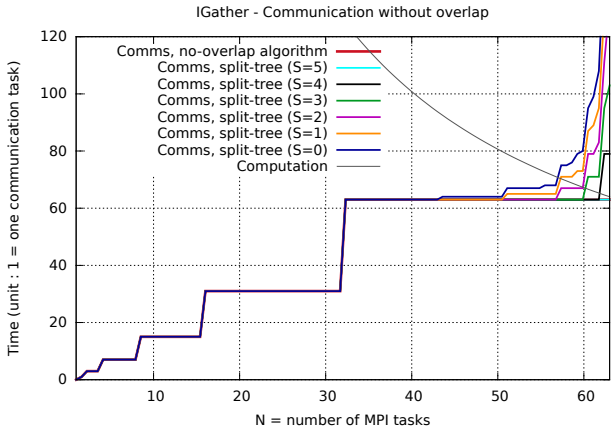


Figure 15. Model of communication cost for operations with increasing buffer size (scatter, gather) on 64 cores.

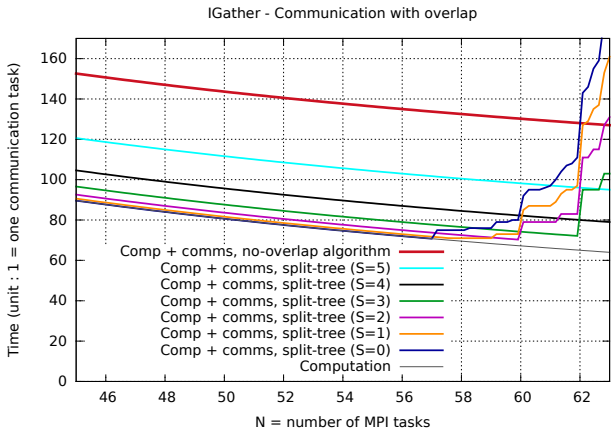


Figure 16. Model of communication/computation overlap (right) for operations with increasing buffer size (scatter, gather) on 64 cores.

imposed by non-blocking communication (synchronization) is negligible compared to actual communication time.

The model does not depend on hardware performance, thus we don't need any sampling phase to calibrate it prior to running the application. The model is implemented as it is in the code to find the optimal S for a given N.

7 Split-Tree Experimental Results

In this Section, we present experimental results of our split-tree algorithm implemented within MPC.

We implemented our own micro-benchmarking tool to evaluate the performance of our algorithm. This tool works similarly to the Intel MPI Benchmarks[10] but with fixed problem size allowing us to have the same computation workload for different number of MPI tasks (strong scaling). We arbitrary set the buffer size to 2MB and sized the computation workload to reach perfect overlap. Then, we reduce the number of MPI tasks while keeping the same global computation workload. Thus, when we have less MPI tasks, the duration of computation increases and more idle cores are available for progress threads. This contributes to decreasing the time of communications and maximize the overlap. When all cores are used by the MPI tasks, they are no cores left for progress threads. In this case, the algorithm

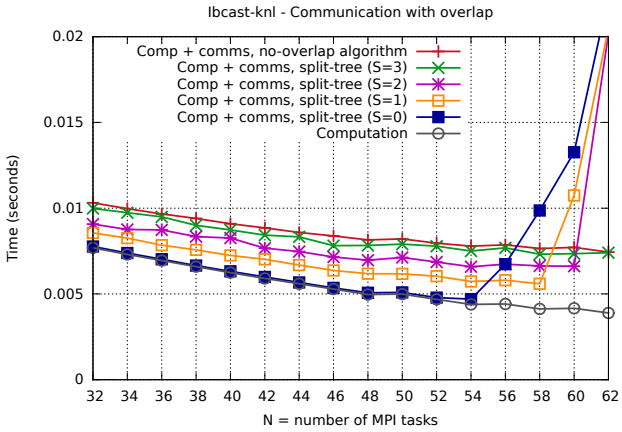


Figure 17. Result of split-tree algorithm with different values of S, for `MPI_Ibcast` with constant-size buffer of 2MB on 64 cores (KNL).

is the same as for the blocking call. Thus we do not show these points in the following performance figures.

We ran our benchmark on two different manycore architectures: a 1.4GHz Intel Xeon Phi Knights Landing with 64 cores (KNL) and a 2.7GHz dual-socket Xeon Platinum Skylake with a total of 48 cores (SKL). The time reported is a median time from several runs. Then, timings from different ranks are aggregated taking into account the slower rank (reduction with max operation).

Comparing split-tree algorithm to default setup. In our first experiments, we tested the interest of the split-tree algorithm. As described in Section 6.3, MPC already provides progress threads for communication collectives. The progress threads are gathered on the available cores. This mapping brings good performances when the number of available cores is high. However, performances collapse when too many progress threads are gathered on the same core. The blue lines labeled "Comp + comms, split-tree (S=0)" show this behavior on KNL for collective Ibcast (Figure 17) and for collective Ireduce on KNL (Figure 18) and Skylake (Figure 19). The label "Comp + comms, split-tree (S=0)" means that no level of the communication tree is done on the MPI tasks, thus all communications are realized on the progress threads.

Thanks to our split tree algorithm, we were able to balance more efficiently communications between the MPI tasks and the progress threads. The orange line labeled "Comp + comms, split-tree (S=1)" (resp. purple line labeled "Comp + comms, split-tree (S=2)" and green line labeled "Comp + comms, split-tree (S=3)") shows the performance of the same algorithms when 1 (resp. 2 and 3) levels of the communication tree remains on the MPI tasks. If enough cores are available to correctly handle the progress threads, the split-tree version is less efficient. This is the expected behavior has the overlap is optimal thanks to the available cores, but some communications are now done on the application cores and can no longer be overlapped. However, when the number of available cores is shrinking, the split-tree version is more stable. For each additional level attached to the MPI tasks, the sudden performance drop is observed with fewer available cores, until S=3 allows to maintain better performances than the blocking call even

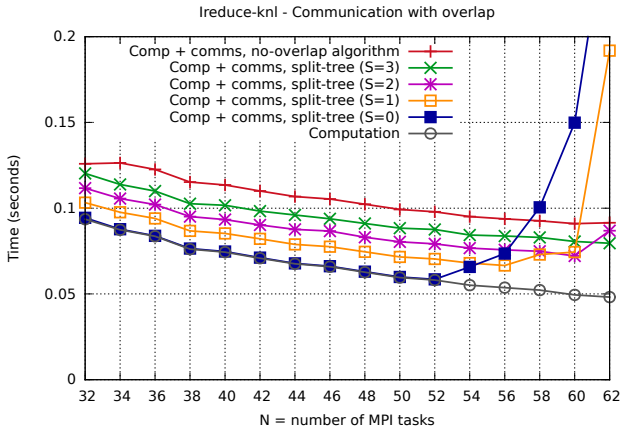


Figure 18. Result of split-tree algorithm with different values of S, for `MPI_Ireduce` with constant-size buffer of 2MB on KNL processor.

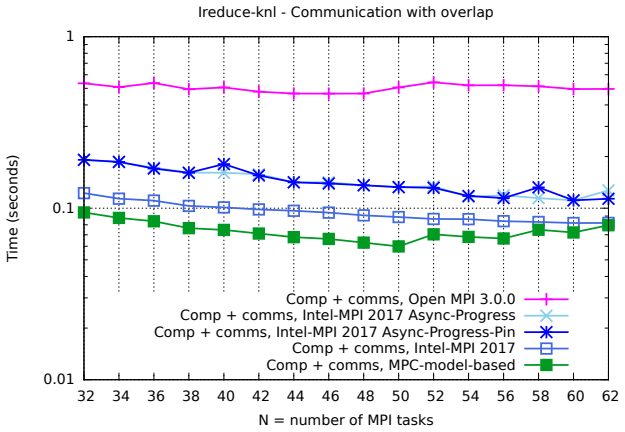


Figure 20. Result of multiple MPI implementation for `MPI_Ireduce` with constant-size buffer of 2MB on KNL processors (Y-axis in log scale).

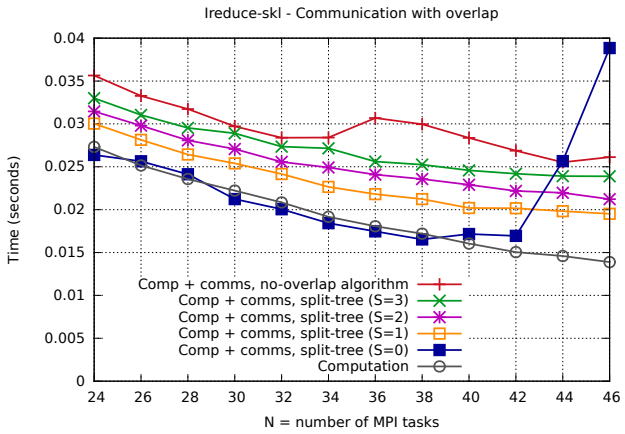


Figure 19. Result of split-tree algorithm with different values of S, for `MPI_Ireduce` with constant-size buffer of 2MB on Skylake processors.

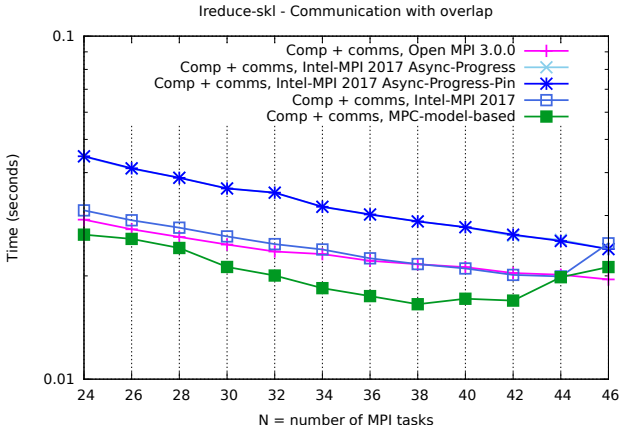


Figure 21. Result of multiple MPI implementation for `MPI_Ireduce` with constant-size buffer of 2MB on Skylake processors (Y-axis in log scale).

in the least favorable case (only one core available for all progress threads). Hence, it is possible to select the best split point S depending on the algorithm and the number of cores hosting progress threads.

Comparing performance results to model. To help select the number of tree levels to leave on the MPI tasks, we proposed a model in Section 6.2. The model projection for `Ireduce` collective on 64 cores is shown in Figure 14. Comparing this projection to the result of `Ireduce` on the 64-core KNL displayed in Figure 18, we can see that the model is really close to the results.

Moreover, the values for switching from a value S in the split-tree to the next one are the same between the prediction and the measured performance. This allows us to select the correct number of levels to leave on the MPI tasks by implementing this model in the MPI runtime system, independently from the performance of the hardware.

On Skylake processor, we have less cores than on KNL. This implies that the depth of the communication tree is lower. In the Figure 19, we show the results for `MPI_Ireduce` on Skylake. We observe a behavior similar to what we observed on KNL.

On the overall, the proposed model is perfect from a qualitative point of view. From a quantitative point of view it also predicts very well the observed behavior despite many simplified assumptions (e.g. no contention, no latency, linear application speed-up).

Comparing MPI implementations. We also compare our algorithm with other MPI implementation such as Intel-MPI and OpenMPI. We ran OpenMPI and Intel-MPI tests with the same compute workload as for our previous experiments. We compare these results to our split-tree algorithm with the S value chosen accordingly to our model. Hence, when the model predicts that an S value is better than another one, this value is automatically applied. For example, on KNL, we switch from $S = 0$ to $S = 1$ for 52 MPI tasks, from $S = 1$ to $S = 2$ for 58 MPI tasks, and from $S = 2$ to $S = 3$ for 62 MPI tasks.

The results for all tested MPI implementation, including our MPC model-based results, are depicted in Figure 20 for KNL and Figure 21 for Skylake with `MPI_Ireduce`.

We observe that our split-tree algorithm, with the selection of the number of levels left on the MPI tasks based on our model (MPC model-based – green), performs well on KNL and Skylake. On KNL, MPC model-based (green lines) is

always better than OpenMPI (purple) and IntelMPI (royalblue). To be fair, we activated for IntelMPI the flags allowing asynchronous progression (*I_MPI_ASYNC_PROGRESS* and and *I_MPI_ASYNC_PROGRESS_PIN*), but these flags reduced the performances (skyblue and blue lines) instead of improving them. On Skylake, OpenMPI performs better than on KNL. However, except for $N = 46$, MPC model-based managed to have better performance thanks to the split-tree algorithm.

Very interestingly, we also see that in this case, the best performance is obtained with 50 cores on the KNL and 38 cores on the SKL, meaning that the best trade-off is far from using all the cores for computation.

8 Conclusion and Future Work

Overlapping communications with computation is the key to amortize the cost of communications, especially for collective communications which are heavier than point-to-point communications. Approaches for progression relying on a progression thread per task suffer from competition between communication and computation and do not actually overlap communication with computation. Approaches relying on a pool of cores dedicated to communication may seem to be a simple solution to this problem; we have shown that collective communications exhibit a slowdown when folded on a few dedicated communication cores.

In this paper, we have proposed an approach to adapt the placement of progress thread to the topology of the collective communication so as to get good overlap and not to slow down the collective operation once folded on the communication cores. We have shown that even the simple chain topology for collective operations does not progress properly with a naive placement on dedicated cores, and have proposed an odd-even placement strategy that obtains good progression.

For tree-based collectives, we have proposed a novel algorithm that combines the best of dedicated communication cores and using application cores. It splits the communication tree so as to execute the narrow part of the tree, representing most of its depth, on dedicated communication cores; this part may be fully overlapped with computation. It places the widest part of the tree, which represents a small part of its depth but a large part of the total work, on all applications cores to benefit from parallelism.

We have modeled the algorithm to demonstrate its relevance and to tune its parameter. We have implemented the algorithm in the MPC software and evaluated its performance on manycore processors (Intel KNL and Skylake). Thanks to the excellent accuracy of the model we are able to almost always find the best trade-off between using dedicated CPU cores or application cores and hence exceed the performance of state-of-the-art competitors. Moreover, it is important to notice that our solution is not bound to the MPC runtime system but can be implemented in any MPI library featuring progress threads for communication.

As future work, we plan to extend the approach of our algorithm to inter-node communications, which have a different behavior than intra-node communications considered in this paper. Moreover, we also plan to improve our algorithm to make it adaptive by detecting the actual amount of computation that may be overlapped instead of assuming it is exactly the same duration as the blocking collective communication. Finally, we will extend autotuning to choose the number of MPI tasks (parameter N) that gets the best overall application performance and not only sections with nonblocking collectives.

We may consider to extend our approach to other collective algorithms, such as pipelined trees. The split-tree algorithm relying on the variability of communicaiton in time, the same approach cannot be applied to pipelined algorithms which are more constant in time. Nevertheless, we may consider to optimize the placement statically using an approach similar to the “odd-even” algorithm.

References

1. Almási G, Heidelberger P, Archer CJ, Martorell X, Erway CC, Moreira JE, Steinmacher-Burow B and Zheng Y (2005) Optimization of MPI Collective Communication on BlueGene/L Systems. In: *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS ’05. New York, NY, USA: ACM. ISBN 1-59593-167-8, pp. 253–262. DOI:10.1145/1088149.1088183. URL http://doi.acm.org/10.1145/1088149.1088183.
2. Coulomb K, Faverge M, Jazeix J, Lagrasse O, Marcoueille J, Noisette P, Redondy A, Thibault S and Vuchener C (2016) Visual Trace Explorer. URL http://vite.gforge.inria.fr/.
3. de Kergommeaux JC and de Oliveira Stein B (2000) Pajé: An extensible environment for visualizing multi-threaded programs executions. In: Bode A, Ludwig T, Karl W and Wismüller R (eds.) *Euro-Par 2000 Parallel Processing*. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-540-44520-3, pp. 133–140.
4. Denis A (2015) pioman: a pthread-based Multithreaded Communication Engine. In: *Euromicro International Conference on Parallel, Distributed and Network-based Processing*. Turku, Finland. URL https://hal.inria.fr/hal-01087775.
5. Derradji S, Palfer-Sollier T, Panziera JP, Poudes A and Wellenreiter F (2015) The BXI Interconnect architecture. In: *High-Performance Inter-connects (HOTI)*. 2015 IEEE 23th Annual Symposium.
6. Hoefler T, Gottschling P and Lumsdaine A (2008) Brief Announcement: Leveraging Non-blocking Collective Communication in High-performance Applications. In: *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures, SPAA’08*. Association for Computing Machinery (ACM). ISBN 978-1-59593-973-9, pp. 113–115. (short paper).
7. Hoefler T and Lumsdaine A (2008) Message Progression in Parallel Computing - To Thread or not to Thread? In: *Proceedings of the 2008 IEEE International Conference on Cluster Computing*. IEEE Computer Society. ISBN 978-1-4244-2640.
8. Hoefler T and Lumsdaine A (2008) Optimizing non-blocking Collective Operations for InfiniBand. In: *Proceedings of the 22nd IEEE International Parallel & Distributed Processing Symposium, CAC’08 Workshop*. ISBN 978-1-4244-1694-3.

9. Hoefler T, Lumsdaine A and Rehm W (2007) Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. In: *Proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07*. IEEE Computer Society/ACM.
10. Intel Coroporation (2018) IMB-NBC benchmarks. https://software.intel.com/fr-fr/node/561946. Accessed: 2018-05-10.
11. Lai P, Balaji P, Thakur R and Panda D (2009) ProOnE: A General Purpose Protocol Onload Engine for Multi- and Many-Core Architectures.
12. Ma T, Bosilca G, Bouteiller A, Goglin B, Squyres JM and Dongarra JJ (2011) Kernel Assisted Collective Intra-node MPI Communication Among Multi-core and Many-core CPUs. In: IEEE (ed.) *40th International Conference on Parallel Processing (ICPP-2011)*. Taipei, Taiwan. DOI: 10.1109/ICPP.2011.29. URL https://hal.inria.fr/inria-00602877.
13. MPI Forum (2012) MPI: A Message-Passing Interface Standard Version 3.0.
14. Pérache M, Jourdren H and Namyst R (2008) MPC: A Unified Parallel Runtime for Clusters of NUMA Machines. In: Springer (ed.) *the 14th International Euro-Par Conference, LNCS*, volume 5168. Las Palmas de Gran Canaria, Spain, pp. 78–88. DOI:10.1007/978-3-540-85451-7_9. URL https://hal.inria.fr/inria-00422229.
15. Rashti MJ and Afsahi A (2008) Improving communication progress and overlap in MPI Rendezvous protocol over RDMA-enabled interconnects. In: *High Performance Computing Systems and Applications, 2008. HPCS 2008. 22nd International Symposium on*. IEEE, pp. 95–101.
16. Ruhela A, Subramoni H, Chakraborty S, Bayatpour M, Kousha P and Panda DK (2018) Efficient asynchronous communication progress for MPI without dedicated resources. In: *Proceedings of the 25th European MPI Users' Group Meeting*, EuroMPI'18. New York, NY, USA: ACM. ISBN 978-1-4503-6492-8, pp. 14:1–14:11. DOI:10.1145/3236367.3236376. URL http://doi.acm.org/10.1145/3236367.3236376.
17. Sanders P and Träff JL (2006) Parallel prefix (scan) algorithms for mpi. In: Mohr B, Träff JL, Worringen J and Dongarra J (eds.) *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-540-39112-8, pp. 49–57.
18. Si M, Peña A, Balaji P, Takagi M and Ishikawa Y (2014) MT-MPI: multithreaded MPI for many-core environments. In: *Proceedings of the International Conference on Supercomputing*. ISBN 978-1-4503-2642-1.
19. Sur S, Jin H, Chai L and Panda D (2006) RDMA read based rendezvous protocol for MPI over InfiniBand: design alternatives and benefits. In: *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM New York, NY, USA, pp. 32–39.

Author Biographies

Alexandre Denis is a research scientist at Inria. Alexandre Denis got his Master degree from École Normale Supérieure de Lyon in 2000 and his PhD in the domain of Grid Computing from University of Rennes in 2003. In 2003-2004, he was a visiting researcher at Vrije Universiteit Amsterdam. Since 2004, he is conducting his research at Inria Bordeaux Sud-Ouest and at the LaBRI laboratory. He works in the domain of high-performance computing, network communications, and MPI. He is focusing currently on communication/computation overlap.

After defending its Ph.D. in July 2012 on the subject of irregular and multithreaded code optimization, Julien Jaeger started working at CEA. He integrated the MPC team, working on the MPC framework which provides its own MPI, OpenMP and pthread implementations on top of a unified user-level thread scheduler. Since January 2019, he is leading the MPC team. Its topics of interest are parallel programming languages and scheduling, with a focus on MPI and tasks, and how to leverage efficient asynchronous execution.

Emmanuel Jeannot is a senior research scientist at Inria. He is conducting his research at Inria Bordeaux Sud-Ouest and at the LaBRI laboratory since 2009. Emmanuel Jeannot got his PhD degree in computer science from Ecole Normale Supérieure de Lyon, at the LIP laboratory in 1999. After his PhD, he spent one year as a postdoc at the LaBRI laboratory in Bordeaux. From 2000 to 2009, he did his research at the Loria Laboratory in Nancy. From 2000 to 2005, he was assistant professor at the Université Henry Poincaré, Nancy. From 2005 to 2009, he worked for the Nancy Grand-Est Inria research centrer. Additionally, in 2006 he was a visiting researcher at the University of Tennessee, ICL laboratory. His main research interests spans the vast domain of parallel and high-performance computing and more precisely: processes placement, topology-aware algorithms, scheduling for heterogeneous environments, data redistribution, I/O and storage, algorithms and models for parallel machines, adaptive online compression and programming models.

Marc Pérache is a research engineer at "Commissariat à l'Energie Atomique et aux Energies Alternatives". He has been working in the HPC field since 2003, starting with a Ph.D. on runtime systems for cluster of NUMA nodes (received in 2006). During his Ph.D., he started the MPC (MultiProcessor Computing) framework. He joined CEA in 2006 as a research engineer to extend the MPC framework and provide building blocks for high performance multithreaded applications. He received the "Habilitation à Diriger des Recherches" (French degree which accredits to supervise researches) from Versailles Saint Quentin-en-Yvelines University in 2015. Since 2018, he leads an R&D group dealing with runtime systems, tools for HPC systems and framework for scientific computing codes.

After passing his master degree on High Performance Computing at University of Versailles in 2015, Hugo Taboada obtains his Ph.D degree from the University of Bordeaux under the direction of Emmanuel Jeannot and Alexandre Denis in 2018, working on MPI Non-blocking collective optimization. He is now working at CEA to help application benefit from architecture and runtime specificities. Its topics of interest are scheduling, thread placement, and communication overlap.