Towards leveraging collective performance with the support of MPI 4.0 features in MPC

Stephane Bouhrour[a], Thibaut Pepin[b], Julien Jaeger[a,b,c]

[a]*Exascale Computing Research Laboratory, 2 rue de la piquetterie, Bruyères-le-châtel, 91680, France*
[b]*CEA, DAM, DIF, Arpajon, F-91297, France*
[c]*Université Paris-Saclay, CEA, Laboratoire en Informatique Haute Performance pour le Calcul et la simulation, Bruyères-le-châtel, 91680, France*

Abstract

Persistent collective communications and communicator splitting according to the underlying hardware topology have recently been voted in the MPI standard. Persistent semantics contains an initialization phase called only once for a specific collective operation, with subsequent recurring invocations. This opens the door to many optimizations requiring heavy setup costs to improve collective performances. Communicator topological splitting offers a standard way to design topological algorithm through the use of sub-communicators mapped to hardware hierarchical levels. Setting these communicators might be too costly to be efficient on a single collective call. However, the persistent semantics allow to create these communicators once at initialization, and use them repeatedly in the multiple collective invocations to have an efficient algorithm.

In this paper, we describe the implementation of these two new MPI features in the MPC framework. We first present a naïve and an optimized version of persistent collectives without topology knowledge. Then, after detailing the implementation of hardware topology splitting and the hierarchical levels supported in MPC, we showcase how these two features can be combined to produce efficient topology-aware persistent collective implementations. Experimental results show that the topology-aware algorithms built with these basic blocks offer good performances, independent of the MPI processes binding.

1. Introduction

To leverage the whole computing power of a supercomputer, High-Performance Computing (HPC) applications run on numerous compute nodes, inducing data exchange between the compute workers on these nodes. To this aim, Message Passing Interface (MPI) was introduced in 1994, and quickly became the *de-facto* standard for internode communications.

The first version of the MPI standard already introduced the concept of blocking, nonblocking and persistent point-to-point operations [1]. The persistent semantics decouples the initialization of the associated communication from its actual execution. A first procedure call initializes the operation. A second call starts and a third call completes the operation. Before freeing the operation, and its associated structures, with a fourth procedure call, it is possible to perform again the same operation multiple times with new calls to the set of starting and completing procedures.

MPI-1 also provided the concept of collective communications. These communications involve a group of MPI processes, which have to participate in a global communication pattern (such as broadcasting a value from one MPI process to all others, or reducing values located on each MPI process). In MPI-1, only blocking collectives were introduced. Nonblocking collectives were added much later in version 3.0 of the standard [2]. Because persistent communications may offer some performance advantage over nonblocking ones, persistent collectives have been studied for more than twenty years [3, 4], eventually leading to its proposal and adoption into the latest version 4.0 of the MPI standard.

Numerous collective implementation optimizations rely on hardware topology awareness. Unfortunately, the MPI standard did not originally provide mechanism to detect the hierarchical topology of the hardware. Hence users wanting to design topological communication patterns have to rely on external tools, such as Hwloc [5]. From this observation, new communicator splitting values were proposed to be added in the newest MPI standard. They allow to split a given communicator in sub-communicators related to the underlying hardware hierarchy.

Building all hierarchical communicators can be cumbersome, and this setup phase cost may be to high to be covered by the performance gain of a unique collective execution. However, the main advantage of persistent communications when compared to nonblocking communications is the possibility to perform the same operation several times without having to initialize it each time. Thanks to this, it is then possible to take more time in the initialization phase of the operation, e.g., to generate a specific topological algorithm with newly created hierarchical communicators, to find the best performing algorithm. If the compute time of performing the operation is reduced, then performing the operation enough times will allow hiding the initialization costs, and provide an overall speedup.

This paper is an extended version of a previous paper published and presented at EuroMPI 2020 [6]. In the previous paper, we focused on the lessons learned while implementing persistent collectives in the MPC framework [7]. In this paper, we

Email addresses: stephane.bouhrour@uvsq.fr (Stephane Bouhrour), thibaut.pepin@cea.fr (Thibaut Pepin), julien.jaeger@cea.fr (Julien Jaeger)

recall the main implementation details of our persistent collectives. We also add the description of our newly implemented support for communicator topological splitting. Then, we describe how we generate topological communicators along with *conduits communicators*, allowing to easily design topology-aware algorithms. Thanks to this method, we create simple, topology-aware algorithms for persistent broadcast, gather and reduce, showing good and stable performances.

The paper is organized as follows. Section 2 presents the related work describing previous work on optimizing collectives, especially through topology concerns. Section 3 describes the persistent communications interface, before recalling the main details of our persistent collective implementation in Section 4. Some experimental results regarding the optimizations in this implementation are also recalled in Section 5. Then, Section 6 describes the communicator topological splitting interface, before detailing our implementation of this feature inside MPC in Section 7. We showcase how to build all necessary communicators to easily design hierarchical algorithms in Section 8, and design very simple algorithms using this feature in Section 9. Section 10 presents experimental results with our topology-aware persistent collectives compared to original algorithms in MPC and OpenMPI, before concluding in Section 11.

2. Related Work

2.1. Topology awareness

Numerous work exists on the optimization of collective communications, especially with a focus on hierarchical collective patterns. Ruefenacht *et al.* [8] perform an extensive study on the recursive doubling algorithm for the Allreduce communication pattern. Hsanov *et al.* [9] analyze different allreduce communication algorithms, such as plain linear, ring, recursive doubling and Rabenseifner's algorithm based on reduce-scatter and allgather communications. For the alltoall communication pattern, Kang *et al.* [10] also develop a new communication algorithm to benefit from the topological hierarchy and locality to target optimal bandwidth. All these works, and numerous others from the last decades [11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22], show the importance of considering the underneath hardware topology to optimize the communications. Yet, the MPI standard, up to version 3.1, did not directly provide an interface to extract or the underlying hardware topology. Users have to create their own software to pin MPI processes to specific resources, such as MPI-pin [23] or mpibind [24]. However, the initial binding may not be enough, as the different communicators in a program may involve different set of resources, hence requiring different bindings. A standard and generic way should be provided to adapt any communication pattern to the underlying topology. This drove the proposal and acceptance of new additions improving the support for the hardware topology in the version 4.0 of the MPI standard, lead by the work of Goglin *et al.* [25].

2.1.1. Caching and persistence

Persistence and caching is a key concept to reduce overheads and increase performances. As an example, Hoefler *et al.* [26] propose optimizations on neighborhood collectives schedules through hints given by the user. They identify three levels of persistent information: persistent topology, persistent message sizes and persistent buffer accesses. Through the user hints, they use this knowledge to improve the neighborhood collective performances. On the same general idea, Träff *et al.* [27] tackle repetitive isomorphic sparse neighborhood collectives. Message-combining communication schedule and MPI datatypes used in their isomorphic communications can be cached to avoid creation costs. They implement their own isomorphic sparse neighborhood persistent collective, creating the cached structure in the initialization phase, allowing them to implement zero-copy `alltoall` and `allgather` operations. Another work using MPI datatypes capabilities to implement a zero-copy all-to-all [28] also presents the benefit of such approach in a persistent collective scheme. A very recent work aiming to provide a more generic and efficient cartesian collective feature for MPI [29] already designed its algorithms in anticipation of a persistent interface.

Optimization of persistent collectives could already be found before their adoption in the MPI standard. Hori *et al.* [30] optimize the use of RDMA engines on the K computer with applications implementing redundant neighborhood communication involving ghost sharing regions. They compute an optimal schedule of communication requests onto multiple RDMA engines for those persistent communications to enhance their latencies and avoid resource contentions as much as possible. In a more recent work, Hatanaka *et al.* [31] implement persistent communications with the MPI generalized request interface. They expose their results on a `neigborhood_alltoallw` using their interface on the FX100 system at RIKEN AICS. While the performance improvements of these implementations are impressive, they are based on a specific underlying hardware. Such optimization is hard to directly integrate into another MPI implementation. However, specializing our persistent collective implementation to benefit from the underlying network capability may help to improve the performances.

Getting the hardware topology information can be time and resource consuming. The performance gain provided by a topological algorithm should absorb the extra-time used for retrieving the topology knowledge. Persistent communications are designed to be called several time in a program life-time, with only one initialization call. Hence, they are perfect candidates to design and embed topological algorithms.

In this paper, we show that persistent initialization calls can be used to gather and cache the hardware knowledge. Using the future MPI communicator splitting interface, the hierarchy of communicators and communication pattern can be build for the input communicator of the persistent collectives. The methodology to design the hierarchy of communicators, and to easily build topological algorithm, can be used to implement the topology optimizations of numerous related work in a more generic way using a standardized interface. We showcase this with an implementation of both features in the MPC Framework.

2.1.2. MPC Framework

The MPC framework is a unified parallel runtime for programming clusters of NUMA machines. MPC provides both its own MPI implementation and OpenMP implementation, on top of a unified user-level thread scheduler. Thanks to its tight coupling with threading models, MPC-MPI offers both a thread-based and a process-based execution scheme, which can be selected at runtime [7]. This tight coupling also allows better handling of inner progression threads. The nonblocking procedures implemented in MPC come from the libNBC [32], which has been adapted to use MPC user-level threads as progress threads.

Having a coherent distribution of MPI processes and OpenMP threads in hybrid programs requires some knowledge about the node hardware topology. MPC relies on Hwloc [5] to gather node topology information, and build its own internal topology structure used for both worker placement and data locality [33].

Thanks to its nonblocking collectives implementation based on a well-known library and its inherent topology awareness, MPC is a good candidate to implement both persistent collectives and communicator topology splitting.

3. Persistent collective interface

A persistent collective operation, as a persistent point-to-point operation, follows the persistent operation semantics. The persistent semantics augments the nonblocking semantics, which needs at least two procedure calls, with two other required procedure calls.

In the case of a nonblocking operation, a first procedure, prefixed with `MPI_I`, initializes the operation, and may return before the operation is completed (or even started). The initialization procedure fills in an `MPI_Request` object used to identify the associated operation. Once the initialization call is done, the operation can be performed at any time by the MPI runtime. To complete the operation, at least one subsequent call to a completion procedure (`MPI_Wait`, `MPI_Test`,...) is necessary, using the `MPI_Request` object to identify the operation to finish. Once the operation is done, the `MPI_Request` object is emptied, and can be reused. A usage example of a nonblocking operation inside a `for` loop is shown in Listing 1, with both initialization and completion procedures having to be invoked inside the loop.

For a persistent operation, at least four MPI procedures are required to perform and complete the operation.

- The first procedure, with the form `MPI_<Coll>_init` (e.g., `MPI_Bcast_init`), initializes the operation, and fill the provided `MPI_Request` object. The input arguments are bound to the `MPI_Request` object and will never change until the request is freed. On the return of the call, the operation cannot start yet.

- A second procedure, `MPI_Start`, starts the persistent operation, i.e., it informs the MPI runtime that the operation can be performed. The `MPI_Request` object is then used for the `MPI_Start` procedure to know which operation can be started and with which parameters.

- As with nonblocking, a completion call is necessary to complete the operation, such as `MPI_Wait`, `MPI_Test` or their `all/some/any` derivatives. Contrary to nonblocking operations, the `MPI_Request` for a persistent communication is not freed after completion. The `MPI_Request` object still contains all information relative to its persistent communication, and it is only marked as `inactive`. As such, the operation identified by the `MPI_Request` object can be performed again, through another call to the `MPI_Start` procedure followed by a completion call. These new executions of the collective will be performed with the exact same parameters, stored in the `MPI_Request` object during the initialization call. It is interesting to notice that, though the pointers to the input and/or output buffers are the same, the content of the buffers are permitted to change between two operation executions (hence after a completion call and before the next call of the starting procedure).

- Once the user is sure to never perform the same operation again, a call to `MPI_Request_free` empties the `MPI_Request` object and finishes the persistent operation. `MPI_Request_free` can only be called on an inactive persistent collective request.

A usage example of a persistent operation inside a `for` loop is shown in Listing 2. Note that this example shows the ideal usage of a persistent operation. Since the operation can be performed several times with only one initialization, only the procedures needed to start and complete the operation are invoked in the loop. Contrary to the nonblocking case, the initialization procedure is called only once, *before* the loop. After the loop ends, the freeing procedure is invoked, marking the `MPI_Request` object as free and being usable for another operation.

4. Persistent collectives implementation

This section describes the implementation of persistent collectives in the MPC framework. We first present a naïve implementation directy based on nonblocking communication calls. Then, we describe the caching optimizations that were performed in our naïve implementation to benefit from the persistent semantics and increase the performances. The first caching optimization consists in caching the *schedule* structure, as done in the libPNBC library [3]. The second optimization offers another caching level targeting deeper internal structures. The details on the nonblocking collectives implementation in MPC, and the internal sctructures that will be touched by these optimizations, can be found in our EuroMPI 2020 paper [6], Section 3.2.

4.1. Naïve implementation of persistent collectives

A first naïve implementation of persistent collectives can be done using nonblocking procedures.

The idea is simple: it consists in creating a `MPI_Request` object, and give it to the `MPI_<Coll>_init` call, as imposed

Listing 1: Nonblocking benchmark implementation

```
MPI_Request req_ptr
double wwtime = 0;
MPI_Barrier();
START_TIMER(wwtime)

for(i=0; i<500;i++){
        MPI_I<coll>(args_coll, &req_ptr);
        MPI_Wait(&req_ptr, status);
}

END_TIMER(wwtime)
double recup_max = 0;
MPI_Reduce(&wwtime,  &recup_max, 1, MPI_DOUBLE,
    MPI_MAX,0, MPI_COMM_WORLD);
```

Listing 2: Persistent benchmark implementation

```
MPI_Request req_ptr
double wwtime = 0;
MPI_Barrier();
START_TIMER(wwtime)
MPI_<Coll>_init(args_coll, &req_ptr);
for(i=0; i<500;i++){
        MPI_Start(&req_ptr);
        MPI_Wait(&req_ptr, status);
}
MPI_Request_free(&req_ptr)
END_TIMER(wwtime)
double recup_max = 0;
MPI_Reduce(&wwtime,  &recup_max, 1, MPI_DOUBLE,
     MPI_MAX,0, MPI_COMM_WORLD);
```

by the MPI standard. This request is tagged as persistent and `inactive`, and gathers all other arguments passed down to the persistent initialization call. In addition, the type of the collective operation, determined by the `Coll` part of the initialization procedure name, also has to be stored.

The next calls to `MPI_Start`, or `MPI_Startall`, function with the `MPI_Request` object are supposed to start the operation. In this naïve implementation, the call retrieves the operation type stored in the object. Once the collective operation type is retrieved and recognized, the corresponding nonblocking operation is started by calling the appropriate procedure (e.g., for a persistent initialized by `MPI_<Coll>_init`, the procedure `MPI_I<coll>` is invoked at this step). All the arguments passed to the persistent initialization call and stored in the `MPI_Request` object are now directly given to the nonblocking initialization call. A new `MPI_Request` object is created to be passed to the nonblocking initialization procedure to represent the nonblocking operation. This new request is embedded in the `MPI_Request` object representing the persistent operation. We then have two requests, a persistent request, which is the `MPI_Request` object representing the persistent operation and embedding a nonblocking request, the new `MPI_Request` object representing the current nonblocking operation. Before leaving this call, the persistent request is set to `active`.

Once the nonblocking operation is completed through the call of a completion call (e.g., `MPI_Wait`, `MPI_Test` and their derivatives), the nonblocking request is freed and can be reused. However, the persistent request must not be destroyed since it represents a persistent operation. It is simply set to be `inactive` so a new operation can be started.

Such an implementation is functional and in accordance with MPI-4 standard. Nevertheless, no benefit is made via the persistent mechanism as, since nonblocking procedures and operations are used, all internal structures needed to build the nonblocking collective are created and initialized at each call, hence each time a new operation is started.

4.2. Caching schedule optimization

The first caching optimization consists in shifting the building cost of the algorithm from the nonblocking collective initialization (performed in the `MPI_Start` call) to the persistent initialization `MPI_<Coll>_init` call. It is very similar to the optimization performed in libPNBC [3]. This optimization level will be referred as "Schedule caching" in Section 5.

In our implementation, the nonblocking collective initialization function is separated into two functions: one function dealing exclusively with the initialization part, which becomes the persistent initialization function, and another internal function used to start the operation and schedule all the point-to-point operations required by the collective algorithm. The modifications are described in the following paragraphs.

Initialization. The call to `MPI_<Coll>_init` stores and references all arguments of the collective inside an internal structure of the MPC runtime. It then realizes all the initialization steps of the corresponding nonblocking collective operation, including the creation of the *schedule* structure depending on the collective and its chosen algorithm.

Once this persistent initialization function is completed, the application resumes and the `MPI_Request` object contains all the information needed to execute the operation.

Starting. Subsequent calls to `MPI_Start` access all the required information referenced by the request object given in argument and call the corresponding collective internal function created by the splitting of the nonblocking initialization procedure.

The *schedule* is retrieved from the request. Then, the necessary variables and internal structures are created and initialized to the right values to track the progression of the algorithm. For example, as MPC relies on pointer arithmetic to progress in the schedule, the initial reading pointer is reset to the beginning of the schedule.

Completion. Upon completion of the current persistent operation, all the internal structures built for the execution of the operation are not released, contrary to the previous naïve version based on nonblocking operations. Keeping the internal structures allow saving the overhead of doing their initialization for each operation.

Freeing. The `MPI_Request` object and all intermediate structures, including the *schedule* structure, are released upon freeing. As described in the MPI standard, only requests marked as *inactive* should be released by a call to the `MPI_Request_free` procedure.

4.3. Internal intermediate persistent requests optimization

This second caching level implements an optimization hinted in Morgan *et al.* [3] when presenting the libPNBC library. In the original version of libNBC, thus also in libPNBC, when parsing the *schedule* structures, the nonblocking operations composing the collective algorithm, and the associated *rounds*, are created. Thus, each time a persistent operation is started, time is spent to initialize the same recurring point-to-point operations.

Another optimization possible for the implementation of the persistent collectives is to create them once at initialization, and retrieve them in the subsequent `MPI_Start` calls. Morgan *et al.* states that such a transformation is difficult to perform in libNBC, thus also in libPNBC, as "that requires invasive code-changes to the whole scheduling storage/usage code". However, in MPC, such rewriting of the *schedules* and *rounds* management was already performed in the integrated version of libNBC to avoid heavy allocating costs. Hence, it is now possible to implement this additional optimization. This new caching optimization will be referred in Section 5 as "Request caching". We argue that this implementation further demonstrates the possibility of persistent collectives, and shows the potential speed-up libPNBC could reach if applying this new caching level.

Initialization. For this new optimization, after creating the *schedule* structure, it is parsed a first time to be filled. The algorithm chosen for the collective determines the point-to-point operations and *rounds* necessary to perform the operation. These operations and *rounds* are then created, and used to fill in the *schedule* structure. By doing so, we save the cost of recreating and inserting those internal structures for each `MPI_Start` call. As the *schedule* is already embedded in the `MPI_Request` object associated with the persistent operation, the corresponding `MPI_Start` call will be able to retrieve all these information and perform the operation.

It is possible that a persistent operation is initialized but never used, or the operation might be performed for the first time long after it was initialized. We decided to to create and embed such structures in the `MPI_Request` object during initialization to avoid unnecessary allocations, memory consumption and depleting available internal structures.

Start. At the first call to `MPI_Start` after the initialization of the persistent collective, the internal requests for the intermediate point-to-point operations are created round after round during the progress, as with nonblocking operations. These internal requests are kept and cached inside the encompassing `MPI_Request` object. For the other calls to `MPI_Start`, for the same persistent operation, the parsing of the schedule do not create internal requests anymore. Instead the point-to-point intermediate requests are retrieved and reinitialized to be able to perform again the same operation.

Due to this recurring behavior, the operations associated with these internal requests can be seen as internal persistent point-to-point requests.

Completion. At the end of each operation, the internal requests are not released anymore. The first time a persistent collective operation is performed, the associated request handle is tagged to indicate that the internal requests for the intermediate point-to-point operations have been built and can be retrieved when starting again this operation.

Freeing. The `MPI_Request` object and all intermediate structures are released upon freeing. These intermediate structures include the *schedule* structure, along with the *rounds*, intermediate point-to-point operations and their associated internal requests. As described in the MPI standard, only requests marked as *inactive* should be released by a call to the `MPI_Request_free` procedure.

5. Persistent collective experimental results

We evaluate our persistent collective implementations, studying the impact of both caching optimizations compared to the naïve implementation. We first detail the experimental setup, before detailing the results.

5.1. Test description

To measure the raw performance of each version, the microbenchmark consists in a for loop with 500 iterations on which the start and the completion of the collectives are called at each iteration, as shown in Listing 2. We measure the time of the whole loop. Each microbenchmark executed 20 times, and we present the median value of these 20 runs.

We run our tests on Intel KNL nodes with 64 cores (2 nodes available in the test machine), and Intel Sandy Bridge nodes composed of two 8-core sockets for a total of 16 cores per nodes (8 nodes available on the test machine).

To make our comparisons of the different implementations, we firstly vary the number of MPI processes and communicate one byte at each collective call. We run the benchmark in full MPI mode, with one MPI process per core. Hence, for 128 MPI processes, the test spans over 2 KNL nodes and 8 Sandy bridge nodes.

Secondly we study the impact of the size of the arguments buffer passed to the collectives on our implementations. We compare the timing on the slower MPI process for each collective, performing a max-reduce on the timing of all MPI processes.

We chose to present results for 4 collectives extensively used in HPC simulations: `MPI_Bcast`, `MPI_Reduce`, `MPI_Alleduce` and `MPI_Alltoall`, and only for the Intel Sandy Bridge machine which provides more nodes. More collective results are available in our EuroMPI 2020 paper [6], with additional collectives on the Intel Sandy Bridge machine, and all collectives on the Intel KNL machine.

Performance of the naïve implementation. We also implemented the microbenchmark with nonblocking calls, as shown in Listing 1. We compared the nonblocking case to our naïve persistent implementation, and both provided similar results for all tested collectives. The figures for these results are also available in our EuroMPI 2020 paper [6].

5.2. Naïve persistent vs optimizations

Once we checked that our naïve persistent implementations are as performant as the nonblocking procedures, we measure the effect of our optimizations on performances.

We compare the naïve version to the first "Schedule caching" optimization, caching only the *schedule* structure creation as in libPNBC, and to the combination of both "Schedule caching" and "Request caching", the caching of the *schedule* structures along with the caching of internal structures (such as *rounds* and internal requests). Since the naïve version performance is similar to the use of nonblocking procedures, any speedup brought by these optimizations advocates for the benefit of (correctly implemented) persistent collectives over nonblocking collectives.

5.2.1. Number of rank variation

We first display these results on the four chosen collectives, with a count argument of 1 integer (4 bytes), for a various number of MPI processes. The results are shown in Figure 1.

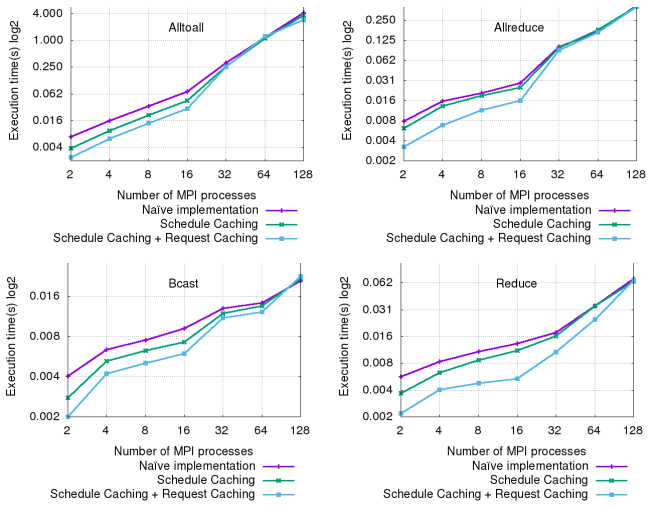

Figure 1: Naïve persistent implementation vs optimizations on Intel Sandy Bridge nodes, varying MPI process numbers with a fixed buffer size of 1 integer (4 bytes)

In all cases, the first optimization (labeled "schedule" with the purple line) provides an improvement over the naïve version (in blue). The second optimizations (labeled "schedule + request") provides an increased speedup compared to the first optimization.

On the other hand, the more the number of MPI processes is important, the less optimizations improvement are noticeable. With a large number of MPI processes, and especially once internode communications begin to appear (over 16 MPI processes on Intel Sandy bridge nodes), the benchmark performance is more and more driven by the communications involved to complete the collective, and less by the building or management costs of internal structures. Since the caching optimization only reduces the overhead on the initialization phase, once the time of the initialization phase becomes negligible compared to the communication time, performance speedup disappears.

The best results are obtained on the reduce collective, with up to a factor of three speedup when performing the collective operation inside a full node (e.g, 16 MPI processes on one Intel Sandy Bridge node). This collective is implemented with a binomial tree. For this algorithm, the critical MPI process is the root, with the greater number of operations to perform of all MPI processes involved in the collective. With such algorithms, most of the intermediate point-to-point operations are performed in parallel, leading to a lesser impact of the communication time, and a greater impact of the initialization and creation costs part. Since these costs are the ones reduced by our caching optimizations, it makes sense that this collective shows the best impact. Bcast and Gather follow the same type of algorithm, but display lesser improvement.

We implemented allreduce, allgather, and reduce_scatter (and there derivatives) as a combination of previous communication patterns (resp. reduce + bcast, gather + bcast and reduce + scatter). Allreduce displays the best speedup of these collectives as it is the combination of two tree algorithms.

For the last collective presented here, alltoall, a lot of communications happen simultaneously, but the critical number of operations is greater than the reduce collective. This mitigates the impact of our caching optimizations.

On all of our test, the scatter collective provides the worst performance. In our implementation, its tree shape is different from Bcast and Gather collectives, which impacts the locality of the data exchanges, hence increasing the average weight of these communications. Thus the impact of the initialization costs, and of our optimizations, is reduced. The complete set of figures can be found in the EuroMPI 2020 paper [6].

5.2.2. Buffer size variation

So far, we tested the collectives for a fixed number of elements to communicate of 1 integer (4 bytes). To be thorough, we also run the benchmark with various buffer sizes. As we saw on the previous Section that internode communications quickly negates the visible effect of the caching optimizations, we decided to test multiple buffer sizes only in intranode context, hence 16 MPI processes on one Intel Sandy Bridge. The results are displayed in Figure 2.

For most collectives, the combination of the two caching levels brings the best performances. This speedup can be seen until the message size is very large (around 2^{16} bytes). With large messages, the creation and initialization costs become negligible compare to the communication time. Thus, the performance gain brought by the optimizations is not visible anymore.

Two collectives display different behaviors: alltoall and, in a lesser extent, allgather. For these collectives, the communication time greatly increases with message sizes above 32 bytes. Because of this, the initialization speedup is reduced, though the alltoall still displays a x1.5 speedup with this reduction.

6. Hardware topological split semantics

The MPI standard has always avoided to make specific reference to the underlying hardware, and especially to the topol-

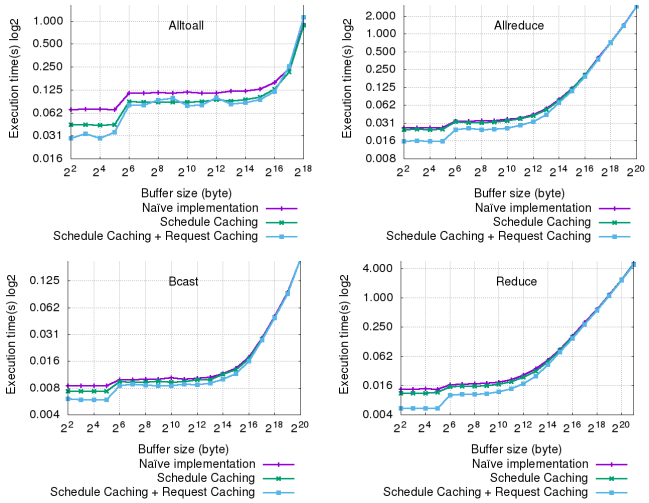

Figure 2: Naïve persistent implementation vs optimizations on one Intel Sandy Bridge nodes, varying buffer sizes with a fixed number of 16 MPI processes (intranode)

ogy. The reason for that was to remain generic and avoid additional burden for the implementations. Even in the latest standard version 3.1, the only reference to hardware topology relates to MPI processes topologies. A user can create a virtual topology for the MPI processes. This topology, attached to a communicator, describes how MPI processes will communicates with each other (MPI process neighborhood), through a multi-dimensional grid or an adjacency graph. Attaching a virtual topology creates a new communicator, in which the user can ask that the MPI processes ranks will be reordered "possibly so as to choose a good embedding of the virtual topology onto the physical machine" (MPI standard 3.1, p.292, l.37-38).

However, as we have shown in Section 2, numerous work related to MPI improvements, specifically for collective communications, focus on hierarchical algorithm to better match the hardware topology. If such optimization can be performed internally by MPI implementation for the collectives pattern defined in the MPI standard, users wanting to design their own communication pattern with topology-awareness had to rely on external tool to gather topology information. For this reason, the new MPI standard version 4.0 provides a way to get topological communicators through specific communicator splitting types.

6.1. Hardware topological split interface

In the MPI standard, communications between MPI processes occur inside a *communicator*. One can think of a communicator as a set of ordered MPI processes (each MPI process as unique id, its rank, only valid in the associated communicator) with a communication context. Different communications happening on different communicators should not interfere with each other a they are in distinct contexts.

One of the procedures proposed in the MPI standard to create new communicators is `MPI_Comm_split_type(MPI_Comm comm, int split_type, int key, MPI_Info info, MPI_Comm *newcomm)`. It allows to split a communicator not on a specific color value, but according to higher-level description represented by the `split_type` argument, optionally accompanied by the `info` argument if necessary. This higher-level description defines which MPI processes should be grouped together in the new communicator. In MPI standard 3.1, only one split_type value is defined: `MPI_COMM_TYPE_SHARED`. Using this split_type value, a user should end up with communicator grouping together MPI processes able to create a common shared memory segment (e.g., through the use of the `MPI_Win_allocate_shared`).

In the new MPI standard 4.0, two new split_type values are defined: `MPI_COMM_TYPE_HW_UNGUIDED` and `MPI_COMM_TYPE_HW_GUIDED`. These two split_type values aim at creating communicators related to the hardware topology.

6.2. Unguided

The goal of the *unguided* split type value is to create topological communicators without any knowledge from the user about the topology levels available. When used with the `MPI_COMM_TYPE_HW_UNGUIDED` split type value, the `MPI_Comm_split_type` procedure should evaluate at which topology level the input communicator corresponds, and return new communicators corresponding to a lower topology level. Saying that a communicator corresponds to a topology level means that every MPI process in the communicator can utilize the specific hardware resource type instance at this level, and that they cannot utilize another instance of this hardware resource type. For example, a communicator corresponds to *L2 cache* topology level if all MPI processes in the communicator have cpusets, hence are linked to compute resources, ensuring that they will always use the same specific L2 cache, and no other L2 cache on the compute node.

The *unguided* split type value follows two rules to create the new topological communicators:

- the new communicators shall always correspond to a topology level strictly lower (i.e., closer to the CPUs) than the original communicator, and,
- the new communicators cardinality shall always be lower than the original communicator (i.e., less MPI processes in the new communicators than in the original one).

These two rules ensure that a user will not create several identical communicators through the *unguided* interface. If such criteria cannot be met, then `MPI_COMM_NULL` is returned. Note that `MPI_COMM_NULL` can be returned for some of the MPI processes in the original communicator, while a subgroup of MPI processes can effectively meet the requirements and get valid non-null communicator in `newcomm`.

A user can create communicators for each level of his hardware topology by calling the `MPI_Comm_split_type` procedure recursively, with `MPI_COMM_TYPE_HW_UNGUIDED` as split type value and using the output communicator `newcomm` as the input communicator for the next split. Doing so, starting with `MPI_COMM_WORLD`, the procedure will first produce communicators corresponding to the first level of the underlying

topology. Then a new call will produce communicators corresponding to the next level in the topology, with strict subsets of the input communicator. Then it will continue until finally reaching a communicator similar to `MPI_COMM_SELF`, then `MPI_COMM_NULL`. Note that the MPI standard does not enforce that an MPI implementation has to support the whole topology hierarchy. It is perfectly valid for an MPI implementation to directly return a communicator similar to `MPI_COMM_SELF` or `MPI_COMM_NULL` when splitting `MPI_COMM_WORLD`.

6.2.1. Guided

The goal of the *guided* split type value it to create topological communicators, with the user specifying to which level the produced communicators should correspond. When used with the `MPI_COMM_TYPE_HW_GUIDED` split type value, the `MPI_Comm_split_type` procedure `info` argument should also be specified. The `mpi_hw_resource_type` info key is reserved in the MPI standard for this specific use. Its only value explicitly specified is `mpi_shared_memory`; it is defined to provide a similar behavior than when using the already existing `MPI_COMM_TYPE_SHARED` split type value.

If no `mpi_hw_resource_type` info key is specified, or its value is not supported by the MPI implementation, the procedure will return `MPI_COMM_NULL`. If the provided info key value is supported, the procedure will create communicators grouping the maximum subset of MPI processes fitting the criteria. It means that if a user requires a guided split at L3 cache level (level supported by the MPI implementation) and 6 MPI processes utilize the same L3 cache, the procedure has to return one communicator grouping these 6 MPI processes. It cannot produce multiple communicators containing only a subset of these MPI processes (e.g., 2 communicators of 3 MPI processes).

7. Hardware topological split implementation

We implemented the new hardware topology split interface in the MPC framework.

Though it is correct for an MPI implementation to directly return `MPI_COMM_SELF` or `MPI_COMM_NULL` when hardware topology split is asked, we decided to support multiple intranode topology levels. We base our possible topology levels on the ones detected and returned by Hwloc. The supported levels in MPC, i.e., values accepted for the `mpi_hw_resource_type` info key, are: "Node", "NUMA node", "Package" (similar to sockets in a compute node), "L3 cache", "L2 cache", "L1 cache", and "Core".

7.1. Guided implementation

MPC implementation of the `MPI_Comm_split_type` procedure internally uses the `MPI_Comm_split` procedure. `MPI_Comm_split` takes a `color` argument, and the MPI processes with the same `color` value are part of the same output sub-communicator. So, when the *Guided* mode is used, the MPC runtime needs to compute a specific `color` for each hardware object corresponding to the required level. This color is then given to `MPI_Comm_split`.

In MPC, the topology objects provided by Hwloc have been extended to abstract the whole compute node, even with multiple OS processes. Inside a compute node, MPC has a unique id for each element of a topological level. I.e., if there are four L3 caches on a compute node, they will be internally numbered from 0 to 3. Moreover, each compute node involved in a multi-node MPC program also has a unique id. I.e. if the computation spans on P nodes, they will be numbered from 0 to P-1.

Thanks to this internal representation, it is straightforward to compute a unique color for any topological level element. If the asked level for the communicator splitting is "Node" (i.e., compute node), the node unique id in MPC is used as color value. If the asked level is below the node level (e.g., L2 cache), the unique node id is concatenated to each element unique id in the node, thus giving a unique color value for each element at the desired level. To find the correct resource id, each MPI process first retrieves its current location, and the Processing Unit (PU) id associated with its location. Then, the runtime evaluates the type of all the PU ancestors in the topology, until finding a type matching the value passed to `mpi_hw_resource_type`.

7.2. Unguided implementation

The `Unguided` mode consists in two steps: 1) finding which level is the next one in the topology then, 2) using the `Guided` mode implementation by providing the found level as the asked level for splitting.

Two cases arise when searching which is the next level to split. In the first case, the input communicator to split spans over multiple nodes. Since the compute node level is the highest level supported in MPC, if multiple nodes are involved, then the splitting will always target the "Node" level. This check is easily done inside the runtime.

In the second case, the input communicator includes MPI processes already belonging to the same node. It is then necessary to find the correct split level inside the node. One MPI process in the input communicator is elected, then computes pairwise its closest ancestor with every other MPI process of the input communicator. This search is similar to the method to find a resource id in the `Guided` mode. For each pair, the two MPI processes find their hardware location (PU), then go up their ancestor tree in the topology, until reaching a common ancestor. Once each pair is done, the common ancestor with the highest hierarchical level designates the hierarchy level that spans over all the concerned MPI processes. For example, consider that the elected MPI process shares the L2 cache with a second MPI process, and only the L3 cache with the other MPI processes. L3 cache is higher in the topology hierarchy, and, in most of nowadays processor architecture, spans over the L2 cache associated with the elected MPI process. Thus the L3 cache also spans over the second MPI process (sharing the same L2 cache as the elected process, hence the L3 cache is the closest common ancestor for the entire group of MPI processes in the input communicator.

Once the global closest common ancestor is found, then the level just below is the topology level we will use to create the new communicators. This level is then given to the `Guided` mode implementation to produce the expected communicators.

8. Building topological communicators with the new API

As we presented, a lot of collective performance improvement relates to topological algorithms. Since the MPI standard didn't provide any mechanisms to detect the underlying topology, users wanted to elaborate topological communication schemes had to rely on external tools (such as inserting Hwloc calls directly in their code).

The advent of hardware topology splitting in MPI offers a standard way to better map a communication pattern to the hardware topology.

8.1. Building a hierarchy of communicators

With the new interface, building a set of hierarchical communicators is quite easy. Recursively calling `MPI_Comm_split_type`, with `MPI_COMM_WORLD` as input to the first call, then using the generated new communicators `newcomm` as the input communicator for the subsequent calls, will produce on the local MPI process a communicator for each supported level in the topology. The actual snippet of code is presented in Listing 3.

Listing 3: Unguided hierarchical communicators creation

```
hwcomm[level_num] = comm_collective;
MPI_Comm_rank(comm_collective, &rank_comm);

/* create topological communicators */
int level_num = 0;
while((hwcomm[level_num] != MPI_COMM_NULL)
&& level_num < level)
{
  res = MPI_Comm_split_type(hwcomm[level_num],
                MPI_COMM_TYPE_HW_UNGUidED,
                vrank,
                MPI_INFO_NULL,
                &hwcomm[level_num+1]);
  level_num++;
}
```

After executing this code, the user ends up with hierarchical communicators. However, it is not yet possible to do a topological broadcast for example. If we have two hierarchical communicators at the "NUMA node" level, we can realize a broadcast inside each NUMA node using the corresponding communicator. But if the user need to broadcast a value to all MPI processes on the two NUMA nodes, then a communicator grouping all MPI processes should be used. This higher-level communicator only sees a group of MPI processes. If the broadcast is done in this communicator, the algorithm used may be oblivious of the topology.

What a user would want is actually for a unique MPI process of each NUMA node to handle this part of the broadcast. It will allow for a two-step topological algorithm: first inter-NUMA node communications between such local *conduits*, then intra-NUMA node communications.

8.2. Building `conduits` communicator

Once all hierarchical communicators are done, creating such *conduits communicator* is rather easy. Listing 4 displays a snippet of code for this purpose.

Listing 4: Unguided conduits communicators creation for different hardware topology levels

```
int level_num = 0;
while((hwcomm[level_num + 1] != MPI_COMM_NULL))
{
    MPI_Comm_rank(hwcomm[level_num+1], &rank_comm);

    if(rank_comm == 0)
        color = 0;
    else
        color = MPI_UNDEFINED;

    MPI_Comm_split(hwcomm[level_num], color,
         vrank, &conduitcomm[level_num]);
    level_num++;
}
```

To create a *conduits communicator* for a given level, an MPI process should be elected as *conduit* on each communicator corresponding to this level. To this end, we will use the communicator of the above level. Indeed, only communicators at a higher level in the hierarchy will see all the MPI processes that will end up in this *conduits communicator*. We will split such communicator to have one communicator with only the elected MPI processes. The splitting occurs following a given value, i.e., all MPI processes with the same `color` will be grouped. Thus, to have a *conduits communicator*, one only has to give the same color value to the elected MPI processes on each sub-communicator. In Listing 4, all the MPI processes with rank 0 in the sub-communicator have been elected. All sub-communicators have their own numbering, starting from 0 to the size of the MPI processed group: we are sure there is only on MPI process with rank 0 in each sub-communicator. All the MPI processes with rank 0 are given the same color value (0 in the example), while all other MPI processes will use the value `MPI_UNDEFINED`. After splitting the higher-level communicator with such colors, we obtain our *conduits communicator*. The usage of `MPI_UNDEFINED` ensured that no other useless communicators will be created with the splitting call. Choosing rank 0 as the elected MPI process is similar to the implementation of the `MPI_Comm_hsplit_with_roots` function in the original MPI proposal [25]. However, another MPI process can be a better choice. If an MPI process is closer to the network card than the rank 0, then it could be of interest to choose this MPI process as local conduit, as to avoid any additional latency when using the network.

Creating such *conduits communicator* for each supported level, we obtain a chain of *conduits communicators* to implement topological algorithms. Figure 3 presents a graphical representation of such hierarchy of communicators. Consider the supported hardware topology levels are "Node" and "NUMA node", for an initial `MPI_COMM_WORLD` grouping 18 MPI processes. This initial communicator spans over 3 nodes, each node composed of 2 NUMA-nodes, each NUMA-node having 3 Processing units (cores). This hardware architecture is depicted with light grey frames in Figure 3. Creating hierarchical communicators from `MPI_COMM_WORLD`, we will have first 3 communicators at node level, then 6 communicators at NUMA-node level (2 communicators per node). These 6 com-

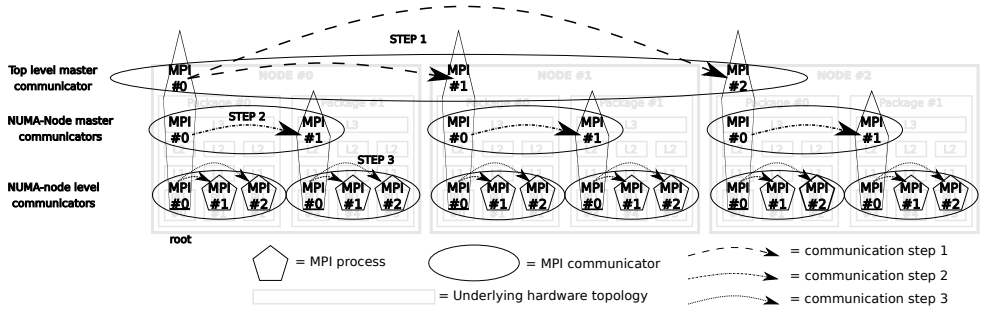

Figure 3: Hierarchy of communicators mapping the underlying hardware. Arrows represent the communication pattern for a topological broadcast algorithm.

municators are depicted with 6 black ellipses at the bottom of the figure, each spanning of 3 cores, and composed of 3 MPI processes numbered from rank 0 to rank 2. To communicate between these 6 NUMA-node communicator, a NUMA-node *conduits communicator* is created per node. These conduits communicators group the elected MPI processes of each NUMA-node communicator (MPI process with rank 0 in this example) per node. Having 3 compute nodes, and 2 NUMA-nodes per node, we end up with 3 NUMA-node *conduits communicators*, grouping 2 MPI processes (one MPI process per NUMA-node communicator). Then, it is necessary to build a top level *conduits communicator*, grouping the elected MPI processes of each NUMA-node *conduits communicators*. Having 3 compute nodes, hence 3 NUMA-node *conduits communicators*, this top level *conduits communicator* regroups 3 elected MPI processes. Note that an MPI process belonging to multiple communicators may have a different rank for each communicator. With such hierarchy of topological and *conduits communicators*, one can easily transfer data following the underlying hardware topology, by using the right communicators in order.

8.3. Discussion on the new MPI 4.0 API

Though the proposed and voted API for the MPI 4.0 standard allows creating topological communicators, additional features can make it easier to use.

Procedure to query supported topology levels. As we described in Section 6, the info key `mpi_hw_resource_type` is defined and reserved to pass at which level the input communicator should be split in `Guided` mode. However, the levels that can be supported, i.e., the possible values for this info key, are completely implementation defined. If such freedom is beneficial for the MPI implementations, it is not user-friendly, as users cannot rely on specific keywords or levels to build their communication pattern. Worse, no procedure is provided to query which levels are supported and could be given to the `Guided` mode. Hence, to use this feature, as user needs to have in-depth knowledge of the utilized MPI implementation to know which levels are possible. Hopefully, such query functions are discussed and envisioned for version 4.1 of the MPI standard.

Creating conduits communicators. As we have seen earlier in this Section, creating the *conduits communicators* to have a useful hierarchy of communicators is straightforward, but cumbersome. The user needs to split each hierarchical communicator with colors related to the rank in the communicator of the previous topology level. If such splitting indeed creates the *conduits communicator* for a specific level, it may also create a communicator with the complementary set of MPI processes, i.e., a communicator regrouping all the MPI processes not elected as *conduit* in the previous level. It is the user's responsibility to directly delete this useless communicator. Otherwise, all the complementary communicators will consume the MPI implementation resources, and could be impairing in the long run. Using the MPI predefined value `MPI_UNDEFINED` will prevent the creation of the complementary communicators.

To alleviate this burden from the user's shoulders and avoid the iterative process to create each *conduits communicators*, a new procedure could be proposed. Taking in input a communicator, the number of levels the use wants in its hierarchy of communicators, and the list of levels to consider, this procedure could return an array of communicators containing directly the *conduits communicators* and the lower-level hierarchical communicator corresponding to the input parameters for each calling process. An info key could drive the selection of the elected MPI process from each communicator level (e.g., MPI process with rank 0 in each communicator, ...).

9. Collective algorithms using hierarchical communicators

In this section, we present how collective pattern can be build from the hierarchical communicators. We focus on broadcast, gather and reduce collective patterns. In the presented algorithm, we make two assumptions: 1) an MPI process elected as *conduit* in a *conduits communicator* will also be the conduit in lower *conduits communicators* and, 2) the root MPI process in a collective is always elected as *conduit*. These two assumptions avoid extra communications (if the root process is not a *conduit*, then a extra step is necessary to either send the data from the *conduit* to the root, or from the root to the *conduit*, depending on the communication pattern).

9.1. Broadcast pattern with hierarchical communicators.

The broadcast collective pattern is very simple. In input, the root MPI process has data, and after the broadcast is executed, all other MPI processes should also have the same data in their respective output buffer.

We design our algorithm for the broadcast collective pattern using hierarchical communicators as follows:

- The root, as local *conduit*, sends the data to all elected MPI processes of the top *conduits communicator* (hence to the elected MPI processes of all compute nodes).
- Now each *conduits* in the top *conduits communicator* has the data. They propagate the data to the other MPI processes they are grouped with the communicator corresponding to the next level in the topology. If the next level is also a *conduits communicator*, the algorithm iterates over this step until reaching the bottom of the communicator hierarchy.
- Once the data has reached the lower topological communicator, the data is broadcast to all local MPI processes, using the elected MPI process at the last hierarchical level as the local root.

In this algorithm, each step realizes an internal broadcast per hierarchical communicators. This internal broadcast can be implemented with any classical algorithm, such as linear of binomial tree.

9.2. Gather pattern with hierarchical communicators.

The gather collective pattern concatenates data from all MPI processes in the output buffer of the root MPI process.

the algorithm for the gather collective pattern is basically the reverse of the broadcast algorithm:
- First, a local gather is done in the lowest communicators. For each such communicator, the MPI process elected as *conduit* for the directly above *conduits communicator* in the topology serves as root.
- Once the local concatenation is finished, another round of gather is done on the directly following *conduits communicators*, and so on until reaching the topmost *conduits communicator*.
- In the topmost *conduits communicator*, a last local gather is done with the root of the whole gather operation as local root.

In the gather operation, the concatenation is done in order with respect to MPI process rank numbering in the input communicator. This can cause some overhead using hierarchical communicators if the ranks are not correctly ordered with respect to the hardware topology.

Figure 4 displays two use cases. In the first case (subfigure (a) on the left), MPI processes are numbered in a contiguous way. Thanks to this numbering, the local gathers at each step already concatenate the data (red squares labelled from "a" to "l") at their rightful place in the output buffer. In the second case (subfigure (b) on the right), MPI processes are numbered in a more random way. This can be the result of a new numbering associated with a virtual topology, or creation of new communicators from merging sparse communicators. It causes the local concatenated buffer to be shuffled compared to the position each data should have in the final buffer. To have the correct output buffer at the end of the operation, it is then necessary to provide data-ownership information. In the displayed algorithm, the rank id associated with each buffer is also gathered at each step (green squares containing the rank id). All rank ids are propagated up to the root on the topmost communicator. Once the root stores locally all data and the associated ownership information, it sorts the buffer to produce the expected result.

It is possible to reduce the size of the ownership information which must be transmitted at each step. A first possibility would be to sort the concatenated buffer at each step. This way, if enough buffers are already at the right position next to each other, their position information can be packed, thus reducing the amount of meta-data to send.

Another way to avoid sending ownership information is to rely on the persistent mechanism. The initialization call for persistent collectives is non-local. Thus, it is possible to exchange data during the initialization step. By gathering in the initialization call the position of all MPI processes involved in the operation, the root MPI process can store locally how the data will be concatenated. The sorting pattern can then be prepared during initialization. When the gather operation will be performed (between `MPI_Start` and the associated completion calls), no meta-data information would need to transit with the actual data.

9.3. Reduce pattern with hierarchical communicators.

The reduce collective pattern is very similar to the gather collective pattern, excepting that is applies an operator on all MPI processes data instead of just concatenating them in the output buffer. However, the ordering of the data may not be as important as for the gather operation. If the operator used in the reduce operation is a commutative operator, then the data can be associated in any order. As such it is possible to use an algorithm which is the reverse of the broadcast algorithm.

This is depicted in the left part of Figure 5:
- First, a local reduce is done in the lowest communicators. For each such communicator, the MPI process elected as *conduit* for the directly above *conduits communicator* in the topology serves as root. This step is depicted in Figure 5 (a) with the dashed-line arrows, marked "communication step 1".
- Once the local reduction is finished, another round of reduce is done on the directly following *conduits communicators*, and so on until reaching the topmost *conduits communicator*. This step is depicted in Figure 5 (a) with the dashed-and-dotted-line arrows, marked "communication step 2".
- In the topmost *conduits communicator*, a last local reduce is done with the root of the whole reduction operation as local root. Then, the root as the result of the whole reduction in its output buffer. This step is depicted in Figure 5 (a) with the dotted-line arrows, marked "communication step 3".

However, if the operator used is not commutative, then the operator should be applied on data in order with respect to the rank numbering. For non-commutative operators, we realize a gather operation (with the same root as the reduction), to obtain the ordered data at the root. Once the root has all the data, it can then apply the operator in order to produce the correct result. This communication pattern is displayed in Figure 5 (b).

9.4. Other collective patterns

We also implemented other collective patterns, based on the three patterns described in the Section. The gatherv operation follows the same algorithm as the gather pattern. Allgather,

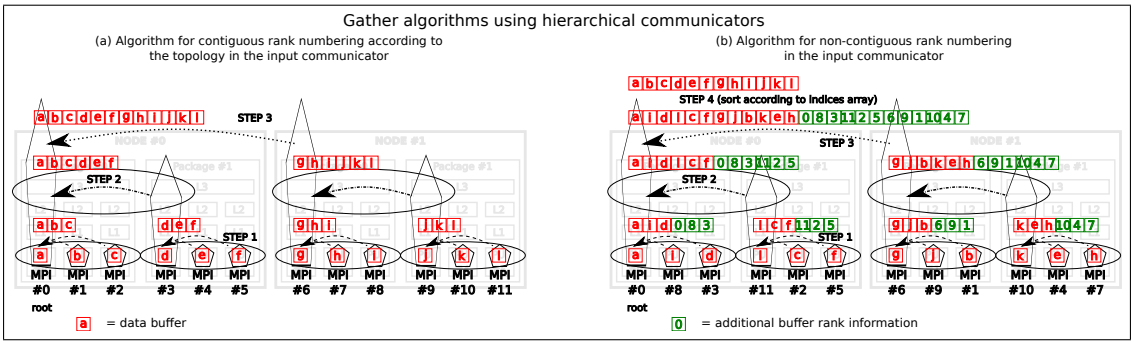

Figure 4: Algorithms for Gather collective using hierarchical communicators.

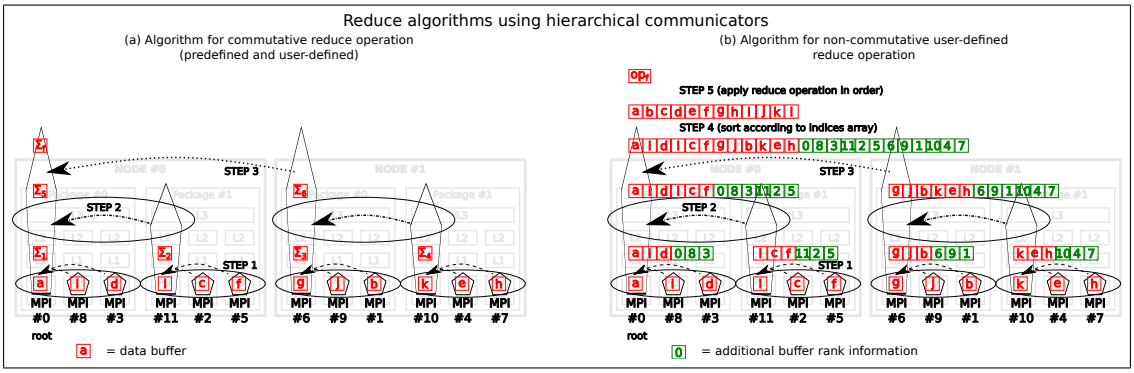

Figure 5: Algorithms for Reduce collective using hierarchical communicators.

allgatherv and allreduce operations are implemented as a combination of gather or reduce operation followed by a broadcast operation, choosing arbitrarily an MPI process as root. Scatter and alltoall operations (along with their derived operations) are currently being designed. We did not consider Scan and Exscan operations so far.

10. Evaluating simple topological algorithm

In this Section, we evaluate the simple algorithms proposed in Section 9. The purpose of these tests is to see if using the new MPI topological splitting MPI allows reaching the performance of algorithms implemented in MPI libraries.

We tested two configurations of hierarchical communicators for our hierarchical collectives. The first configuration has only one level of hierarchical communicators at compute node level, with one *conduits communicator* at the top and topological communicators at the bottom. The second configuration adds a second communicator splitting at the NUMA node level. This second configuration is the one represented in Figure 3. As both configurations present similar performance, we will present only results for 1-level splitting.

For each tested collective, the implementation build the complete hierarchy of communicators (including *conduits communicators*) during initialization. The reduce algorithm is implemented using the gather algorithm (as displayed in Figure 5 (b)). For the gather and reduce algorithms, the position of each buffer is collected at initialization to avoid sending additional meta-data during the actual execution of the collective.

Tests are performed with the same configurations described in Section 5. Each test executes the collective (`MPI_Start` + `MPI_Wait`) 500 times, with only one initialization and one freeing call. The time of the longest iteration is captured. We realize 20 runs for every test, and display the median time of these 20 runs. The experiments were executed on an Intel Sandy Bridge platform, with up to ten nodes.

For each tested persistent collective with MPC, we also measure the corresponding nonblocking collective with OpenMPI 4.0.5 (the more recent version installed in our test machine).

10.1. default vs topological communicators

We compare MPC default algorithm (binomial tree for all tested persistent collectives) to the simple algorithms based on the topological communicators. Figure 6 displays results for gather, reduce, broadcast and allreduce persistent collectives execution, for 1 to 10 nodes filled with MPI processes. The default algorithm is displayed with red curves, while the topology-aware implementation is displayed with green curves. For each algorithm, we test two configurations: ordered ranking (plain lines), and unordered ranking (dotted lines, with "& shuffle" label). The unordered ranking follows a scatter policy. The same "unordering" is applied for all collective test.

For all collectives, we can see that the topological algorithms display similar performances then the binomial algorithm, for an ordered numbering of the MPI processes. However, for the unordered ranking, the binomial algorithm does not perform well. We observe a slowdown of a factor five on broadcast,

and a factor of nine on allreduce. On the other hand, the topological algorithms offer the same performance level for ordered and unordered rank numbering.

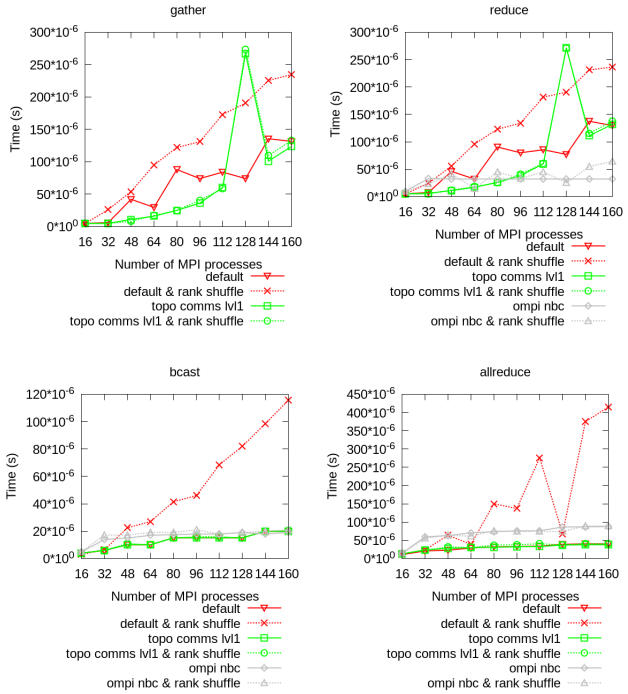

Figure 6: Naïve topological implementation vs default algorithm and default OpenMPI nonblocking algorithm, on Intel Sandy Bridge nodes, varying MPI process numbers with a fixed buffer size of 1 integer (4 bytes)

10.2. topological communicators vs default OpenMPI

We also compared our simple persistent topological algorithms to OpenMPI executions using the nonblocking counterpart procedure. The OpenMPI results are displayed with grey lines, and using the same convention for ordered ranking (plain lines) and unordered ranking (dotted lines).

For the gather operation, OpenMPI was greatly slower than MPC, flattening all the other curves in the graph. We chose to not display these results, as it allows keeping the difference of behavior between the MPC default algorithm and the topological algorithm. For the other displayed collective, the nonblocking default algorithms from OpenMPI offers the same performance for ordered and unordered rank numbering, and is slightly slower that the simple topological algorithms. Our topological gather performs worse than the default behavior because of the time spent in the re-ordering at the last level. This sort is implemented with a very basic algorithm, not scalable. The increasing overhead is due to the increasing number of MPI processes, and thus the increasing number of data to sort. We aim to replace it with a scalable sorting algorithm in the future. The topological reduce follows the same trend, as it has been implemented following the gather algorithm.

Trying to have fair comparison, we also tested the binomial algorithm for the broadcast collective thanks to the *MCA* environment variables proposed by OpenMPI, with the same results. Finally, we tested the blocking versions of these collectives with OpenMPI. The results were also very stable, for both ordered and unordered ranking, but with better performances than the nonblocking version, allowing it to be faster than our topological algorithm.

Nonetheless, these results show that simple algorithms designed using the hierarchical communicators compete with algorithms used in production MPI libraries. Of course, it is still better for a user to utilize the collective algorithms provided by the MPI implementations if its communication pattern fits. But, if the user needs to design a new complex collective communication pattern, the topology-awareness provided by the new API can help design a scalable and efficient implementation, even with unordered rank numbering.

11. Conclusion and future work

Persistent collectives procedures and communicator topological splitting are new features available in the latest MPI standard. In this paper, we present our implementation of both features in the MPC framework. We first detailed naïve implementation for the persistent collectives, then two caching optimizations bringing better performances (mainly in intranode). Then, we described our support of the new communicator splitting interface, with the list of hardware topology levels recognized in our implementation.

We took advantage of the persistent semantics to design simple topology-aware algorithms for broadcast, gather and reduce operations, thanks to the new communicator splitting interface. After describing a methodology to build a hierarchy of communicators and *master communicators* allowing easy topological communication pattern, we used this methodology to build said communicators, and the associated communication pattern, during the persistent initialization call. The induced overhead can then be hidden by the time gained during executions of the persistent collective.

We compared the performance of these new topology-aware persistent broadcast algorithms against our previous binomial persistent implementations. The topology-aware algorithms show similar performance than binomial algorithms for ordered MPI processes rank numbering, but show its benefit with unordered ranking, providing the same performances whereas the binomial algorithms performances dropped. We also compared our simple algorithms to OpenMPI blocking and nonblocking counterparts operations. Results show that these simple algorithms build from the hierarchical communicators compete with algorithms implemented in production MPI library.

We argue that it advocates for simpler collective pattern implementations for MPI users, it the pattern they need is not provided by MPI. Implementing such pattern using persistent semantics allows grouping the cost of hierarchical communicators creation in the initialization phases, which can be hidden thanks to a repetitive call to the more efficient communication pattern.

We discussed the new API for communicator topology splitting, exposing some lacks in the current proposal. We aim to prototype and propose additions to this API to facilitate the creation of *conduits communicators* So far, the algorithms based

on the hierarchy of communicators are available only for persistent collective in MPC, due to the high cost of generating said hierarchy of communicators. However, these algorithms could also be beneficial for blocking and nonblocking collective. We aim at creating a caching system to register a created hierarchy of communicators to the input communicator of persistent collectives. This way, if the same communicator is later used for a blocking or a nonblocking collective operation, this operation can benefit from the attached hierarchy of communicators and directly used the corresponding topology-aware algorithm.

Acknowledgments:. This work was performed under the Exascale Computing Research collaboration, with the support of CEA and UVSQ; it has received funding from the European Unions Horizon 2020/EuroHPC research and innovation programme under grant agreement No 955606 (DEEP-SEA).

References

[1] M. Forum, MPI: A message-passing interface standard, version 1.1 (1994).

[2] M. Forum, MPI: A message-passing interface standard, version 3.0 (2012).

[3] B. Morgan, D. J. Holmes, A. Skjellum, P. Bangalore, S. Sridharan, Planning for performance: Persistent collective operations for MPI, in: Proceedings of the 24th European MPI Users' Group Meeting, EuroMPI '17, ACM, New York, NY, USA, 2017. doi:10.1145/3127024.3127028.

[4] A. Skjellum, High performance MPI: Extending the message passing interface for higher performance and higher predictability, in: International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA'98, 1998, pp. 25–32.

[5] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, R. Namyst, hwloc: A generic framework for managing hardware affinities in HPC applications, in: 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, 2010, pp. 180–186. doi:10.1109/PDP.2010.67.

[6] S. Bouhrour, J. Jaeger, Implementation and performance evaluation of MPI persistent collectives in MPC: a case study, in: W. Bland, K. Mohror, T. Pena (Eds.), EuroMPI/USA '20: 27th European MPI Users' Group Meeting, Virtual Meeting, Austin, TX, USA, September 21-24, 2020, ACM, 2020, pp. 51–60. doi:10.1145/3416315.3416321.

[7] M. Pérache, P. Carribault, H. Jourdren, MPC-MPI: An MPI implementation reducing the overall memory consumption, in: Proceedings of the 16th European PVM/MPI Users' Group Meeting (EuroPVM/MPI 2009), Vol. 5759 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2009, pp. 94–103. doi:10.1007/978-3-642-03770-2_16.

[8] M. Ruefenacht, M. Bull, S. Booth, Generalisation of recursive doubling for allreduce: Now with simulation, Parallel Computing 69 (2017) 24 – 44. doi:https://doi.org/10.1016/j.parco.2017.08.004.

[9] K. Hasanov, A. Lastovetsky, Hierarchical redesign of classic MPI reduction algorithms, The Journal of Supercomputing 73 (2) (2017) 713–725.

[10] Q. Kang, J. Träff, R. Al-Bahrani, A. Agrawal, A. Choudhary, W.-k. Liao, Scalable algorithms for mpi intergroup allgather and allgatherv, Parallel Computing 85 (2019) 220–230. doi:10.1016/j.parco.2019.04.015.

[11] P. Sanders, J. L. Träff, The hierarchical factor algorithm for all-to-all communication, in: Euro-Par 2002 Parallel Processing, Springer Berlin Heidelberg, pp. 799–803.

[12] J. Träff, Hierarchical gather/scatter algorithms with graceful degradation, in: 18th International Parallel and Distributed Processing Symposium, 2004. Proceedings., pp. 80–90. doi:10.1109/IPDPS.2004.1303019.

[13] J. L. Träff, Efficient allgather for regular smp-clusters, in: Recent Advances in Parallel Virtual Machine and Message Passing Interface, Springer Berlin Heidelberg, 2006, pp. 58–65.

[14] J. Träff, Smp-aware message passing programming, in: Proceedings International Parallel and Distributed Processing Symposium, 2003. doi:10.1109/IPDPS.2003.1213253.

[15] N. Karonis, B. de Supinski, I. Foster, W. Gropp, E. Lusk, J. Bresnahan, Exploiting hierarchy in parallel computer networks to optimize collective operation performance, in: Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000, pp. 377–384. doi:10.1109/IPDPS.2000.846009.

[16] N. T. Karonis, B. Toonen, I. Foster, Mpich-g2: A grid-enabled implementation of the message passing interface, J. Parallel Distrib. Comput. 63 (2003) 551–563. doi:10.1016/S0743-7315(03)00002-9.

[17] J. L. Träff, A. Ripke, An optimal broadcast algorithm adapted to SMP clusters, in: B. D. Martino, D. Kranzlmüller, J. J. Dongarra (Eds.), Recent Advances in Parallel Virtual Machine and Message Passing Interface, 12th European PVM/MPI Users' Group Meeting, Sorrento, Italy, September 18-21, 2005, Proceedings, Vol. 3666 of Lecture Notes in Computer Science, Springer, 2005, pp. 48–56. doi:10.1007/11557265_11.

[18] S. Pickartz, C. Clauss, S. Lankes, A. Monti, Enabling hierarchy-aware MPI collectives in dynamically changing topologies, in: Proceedings of the 24th European MPI Users' Group Meeting, EuroMPI '17, ACM, New York, NY, USA, 2017. doi:10.1145/3127024.3127031.

[19] T. Ma, G. Bosilca, A. Bouteiller, J. J. Dongarra, Kernel-assisted and topology-aware MPI collective communications on multicore/many-core platforms, Journal of Parallel and Distributed Computing 73 (7) (2013) 1000–1010. doi:10.1016/j.jpdc.2013.01.015.

[20] Y. Tsujita, A. Hori, T. Kameyama, A. Uno, F. Shoji, Y. Ishikawa, Improving collective MPI-io using topology-aware stepwise data aggregation with i/o throttling, in: Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region, HPC Asia 2018, ACM, New York, NY, USA, 2018, pp. 12–23.

[21] A. Jocksch, N. Ohana, E. Lanti, V. Karakasis, L. Villard, Towards an optimal allreduce communication in message-passing systems, in: EuroMPI/USA, 2020.

[22] A. Bienz, L. Olson, W. Gropp, Node-aware improvements to allreduce, in: 2019 IEEE/ACM Workshop on Exascale MPI (ExaMPI), 2019, pp. 19–28. doi:10.1109/ExaMPI49596.2019.00008.

[23] B. Gerofi, R. Riesen, Y. Ishikawa, Making the case for portable MPI process pinning, EuroMPI '18 posters, 2018.

[24] E. A. León, Mpibind: A memory-centric affinity algorithm for hybrid applications, in: Proceedings of the International Symposium on Memory Systems, MEMSYS '17, ACM, New York, NY, USA, 2017, pp. 262–264. doi:10.1145/3132402.3132415.

[25] B. Goglin, E. Jeannot, F. Mansouri, G. Mercier, Hardware topology management in MPI applications through hierarchical communicators, Parallel Computing 76 (2018) 70 – 90. doi:10.1016/j.parco.2018.05.006.

[26] T. Hoefler, T. Schneider, Optimization principles for collective neighborhood communications, 2012, pp. 98:1–98:10. doi:10.1109/SC.2012.86.

[27] J. Larsson Träff, A. Carpen-Amarie, S. Hunold, A. Rougier, Message-Combining Algorithms for Isomorphic, Sparse Collective Communication, 2016.

[28] J. L. Träff, A. Rougier, S. Hunold, Implementing a classic: Zero-copy all-to-all communication with mpi datatypes, in: Proceedings of the 28th ACM International Conference on Supercomputing, ICS'14, ACM, New York, NY, USA, 2014, pp. 135–144. doi:10.1145/2597652.2597662.

[29] J. L. Träff, S. Hunold, Cartesian collective communication, in: Proceedings of the 48th International Conference on Parallel Processing, ICPP 2019, ACM, New York, NY, USA, 2019. doi:10.1145/3337821.3337848.

[30] M. Hatanaka, A. Hori, Y. Ishikawa, Optimization of MPI persistent communication, in: Proceedings of the 20th European MPI Users' Group Meeting, EuroMPI '13, ACM, New York, NY, USA, 2013, pp. 79–84. doi:10.1145/2488551.2488566.

[31] M. Hatanaka, M. Takagi, A. Hori, Y. Ishikawa, Offloaded MPI persistent collectives using persistent generalized request interface, in: Proceedings of the 24th European MPI Users' Group Meeting, EuroMPI '17, ACM, New York, NY, USA, 2017. doi:10.1145/3127024.3127029.

[32] T. Hoefler, A. Lumsdaine, Design, Implementation, and Usage of LibNBC, School of Informatics, 2006.

[33] M. Tchiboukdjian, P. Carribault, M. Perache, Hierarchical local storage: Exploiting flexible user-data sharing between MPI tasks, in: Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International, pp. 366–377. doi:10.1109/IPDPS.2012.42.