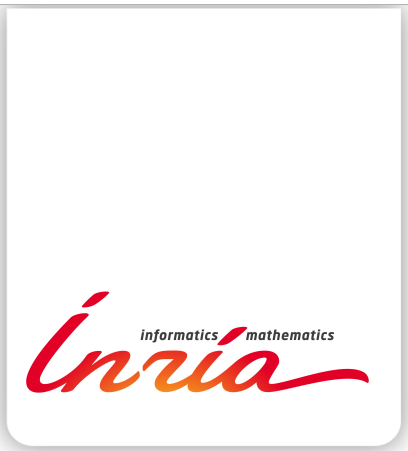
A Hierarchical Model to Manage Hardware Topology in MPI Applications

Emmanuel Jeannot, Farouk Mansouri, Guillaume Mercier

RESEARCH REPORT
N° 9077
June 2017
Project-Team Tadaam

ISSN 0249-6399 ISRN INRIA/RR--9077--FR+ENG



A Hierarchical Model to Manage Hardware Topology in MPI Applications

Emmanuel Jeannot*, Farouk Mansouri†, Guillaume Mercier‡

Project-Team Tadaam

Research Report n° 9077 — June 2017 — 23 pages

Abstract: The MPI standard is a major contribution in the landscape of parallel programming. Since its inception in the mid 90's it has ensured portability and performance for parallel applications on a wide spectrum of machines and architectures. With the advent of multicore machines, understanding and taking into account the underlying physical topology and memory hierarchy as become of paramount importance. The MPI standard in its current state, however, and despite recent evolutions is still unable to offer mechanisms to achieve this. In this paper, we detail several additions to the standard that give the user tools to address the hardware topology and data locality issues while improving application performance.

Key-words: Hierarchy, Hardware Topology, Message Passing

* Inria/LaBRI
† Inria/LaBRI
‡ Bordeaux INP/LaBRI

RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

Un modèle hiérarchique pour la gestion de la topologie dans les applications MPI

Résumé : Le standard MPI est une contribution importante dans le domaine de la programmation parallèle. Il est destiné à l'écriture d'applications parallèles pour un large éventail d'architectures parallèles. L'arrivée des machines multicœur implique une compréhension plus fine de la topologie matérielle sous-jacente, notamment en ce qui concerne les hiérarchies mémoire et réseau. Or, dans son statut actuel, MPI ne permet pas de prendre ces aspects en compte. Nous détaillons dans cet article des modifications à MPI pour permettre la prise en compte de ces aspects afin d'améliorer les performances applicatives.

Mots-clés : Hiérarchie, Topologie Matérielle, Passage de Messages

1 Introduction

Parallelizing or writing from scratch a parallel application is a very challenging task and this challenge has become even more bigger due to the current trend in processors design and supercomputers architecture. Indeed, the hardawre that the programmer has to tackle becomes more and more hierarchically organized. For instance, CPUs now feature various levels of memories that have different properties in terms of size, performance ansd even nature. As a consequence, a parallel application performance is likely to be impacted by the communication occuring between processes and by the way they access data. Thus, if a process accesses some data located in a memory bank physically far from the core it currently executes on, a penalty shall occur (NUMA effect). Also, if two processes share a cache level, they will communicate one with the other more efficiently. This is known as the *data locality issue*.

In order to better exploit the underlying hardware, applications need to take this locality phenomena into account. They need to get a better grasp of the underlying physical architecture they are running on. The current success of `hwloc` [2] demonstrates the need for such a tool and the information it allows to gather. This knowledge may offer the possibility of optimize the code, to better exploit the memory hierarchy or the network topology as well as a global and comprehensive view of the hardware (a continuum of hierarchies between the network and the memory).

To this end, a relevent programming model (along with tools and libraries that implement it) can be of valuable help. The most obvious and natural choice would be to first look at what current parallel programming standards and libraries offer in this area.

In the case of the Message Passing Interface (MPI) [15], some mechanisms are already available that could make it possible for an application to exploit the hardware hierarchy to improve interprocess communication and locality. For instance, in the case of virtual topology management routines, such as `MPI_Dist_graph_create` or `MPI_Graph_map`, the `reorder` argument can be used to create a topology where processes are reorganized according to the underlying physical topology of the target architecture.

However, there are several issues with this approach: first, this argument *might* be used in this fashion, but that is not necessarily the case, which means that the expected behaviour is totally implementation-dependent (and thus not standard) hence likely to change from one implementation to the other or even worse from one implementation version to the other. An application cannot rely on a particular MPI implementation version to be able to use the specific features it needs. Then, in absence of dedicated and relevant mechanisms directly within the standard, an application is forced to use some side-effects of already available features, which constitutes an issue in terms of both interface expressiveness and usability.

In this paper, we present abstraction mechanisms that can help programmers to structure their applications based on physical topology criteria. Such structure can then be used to improve data locality or communication performance by taking into account information that would be otherwise unavailable to the underlying MPI implementation. We also present the implementation of this abstraction and mechanisms in the context of the MPI standard.

This paper is organized as follows: the current status of the MPI interface with regards to hardware topology management is discussed in Section 2 and our proposal is detailed in Section 3. Section 4 describes the target applications of this work and experimental results are analyzed in Section 5, while Section 6 concludes this paper and give potential future directions.

2 Hardware topology management and the MPI Standard

In this section, we shall examine the current possibilities offered by the MPI standard to deal with the issue of hardware topology management in parallel applications. One key-characteristic of the MPI standard is its hardware-agnosticity. Indeed, it makes no assumptions about the hardware on which the application is going to be deployed and run. This behaviour ensures the portability of parallel programms using MPI. It is to be noted also that despite this independence from any hardware considerations in its programming model, the MPI standard and programming model does not prevent from accessing the hardware topology directly from an application.

2.1 Interactions with external tools

One way to tackle this issue is precisely to use an altogether different tool or interface, fully external to MPI. That is, one current practice is to deal with tools representing the hardware topology such as hwloc [2], LibNuma [12], or Pthread sched [16]. These libraries could give a relevent representation of the hardware structure and components. However, they are rather low-level tools and need a good knowledge of the underlying hardware to be used correctly and efficiently. In addition, this practice increases the complexity of developing and supporting codes. Last, as these tools or interfaces are not standard, portability is not guaranteed.

2.2 Current status in the MPI standard and its implementations

The MPI library even offers some means to better understand the nature of the physical architecture in order to exploit it to its full potential. For instance, the extension of Remote Memory Operations with Shared Memory operations in the standard has allowed programmers to structure their applications to take into account the fact that processes are located on the same machine. This can be seen as an alternative to the use of multithreading when memory consumption is at stake [20, Chapter 16]. In this particular case, the MPI standard acknowledges that some physical resource (e.g. memory) is shareable between processes and offers the tools to actually access this resource.

However, despite the presence of some mechanisms in MPI to better understand and use the hardware, they address the issue only partially. Some MPI *implementations* offer mechanisms but since they are implementation-dependent they are typically non-standard. Portability is therefore not guaranteed. The bottom line is: an application should not (or cannot) rely on a specific *implementation* nor on a specific *version* of an implementation (which is even worse). On the other hand, improving performances and scalability of applications is more efficient when locality is exposed during their design steps rather than only relying on MPI implementation optimizations. Thus, the MPI programming model as specified by the standard needs to offer high-level abstractions to take into consideration architecture topology at an early stage and before calling implementations features. The following paragraphs describe some of these mechanisms and their shortcomings.

2.2.1 Shared Memory constructs

`MPI_Comm_split_type` can accept `MPI_COMM_TYPE_SHARED` as a value for its `split_type` argument. The outcome is a communicator that encompasses all processes in the original communicator that can create a shared memory region[1] By using the communicators produced by this function, it is possible to structure an application in order take into account the fact that memory

[1] It is obviously the case when they are located on the same machine, but could also be the case if the processes are on different machines linked by a network *à la* SCI.

is shared: for instance direct memory accesses are possible instead of relying on message-passing exchanges handled by the MPI library. The overheads of the library in case of shared-memory are eliminated. However, the various levels of cache, a major part of the hardware hierarchy, are left unexploited. As explained previously, this memory hierarchy is becoming more and more complex and performance gains are expected from a relevent exploitation of it.

2.2.2 Process topologies and reordering

The various process topologies available in MPI can help to structure an application but they are virtual topologies and quoting the standard itself: "The virtual topology can be exploited by the system in the assignment of processes to physical processor, if this helps to improve the communication performance on a given machine. How this mapping is done, however, is outside the scope of MPI." Some implementations use the `reorder` parameter of some functions (e.g. `MPI_Graph_map`, `MPI_Dist_graph_create` [8], etc.) and take the opportunity to retrieve hardware information and make use of it [14, 19, 9]. Some others tailor the topology routine of MPI to a specific hardware [5, 10]. This, of course, falls into the non-standard category as it is implementation-dependent and is a side-effect of the function. It is not the primary goal of it and the fact that the underlying hardware is efficiently exploited can be seen as a bonus.

2.2.3 Process Managers and process mapping

Process managers can also be of help when it comes to exploit the underlying hardware. Indeed, through their process mapping and binding options, they can allow the user to finely control the way the various MPI application processes are dispatched and executed [7]. Thanks to an adequate placement policy enforced by both these mapping and binding parameters, it is possible to take into account the physical topology and reduce the communication costs for instance [13, 1]. This is also used to improve collective communication performance [22], Unfortunately, these options are totally non-standard and even change from one version of a process manager to the other. This point is outside the scope of this paper, but standardizing these mapping and binding options/policies would ease the user when it comes to launch their application. However, there are tight interactions between mapping/binding polices and the mechanisms proposed in this paper: in absence of a (relevent) mapping/binding of processes, the performance improvements of taking into account the hardware topology are not expected to be as high as if an efficient policy was to be enforced.

2.2.4 MPI Sessions

MPI sessions are a new concept that is currently being discussed by the MPI Forum. In its current state, the standard only allows to call `MPI_Init` and `MPI_Finalize` a single time in an application. This can raise some issues when multiple libraries that internally rely on MPI are used concurrently. Sessions can be seen a lightweight construct, even lighter than groups. A session encompasses MPI processes and some information can be attached to it, for instance about the application or the hardware. In this case, the sharing of hardware resources could be exploited by the application with several different MPI sessions.

3 Proposed Extensions to the MPI standard

As exposed in the previous section, there are currently no means in the MPI standard to portably take into account the hardware topology at the application level. We think that it is important/necessary to offer high-level abstractions helping programmers to take care of locality and

communication optimizations when they design their applications. This way we anticipate and facilitate an implementation work that optimizes communications according to the target architecture. Therefore, we propose to extend the MPI standard and detail in the rest of this section the relevant mechanisms and features needed to achieve these objectives.

3.1 Guidelines

Since its first version in 1994, the MPI standard has grown steadily in terms of number of available routines and functionalities. We therefore advocate for a minimal amount of changes and prefer to leverage existing mechanisms. We prefer not introducing new functions unless it is unavoidable and rather expand existing mechanisms. MPI is about communications and how they are managed. In this regard, exploiting the underlying topology boils down to be able to organize the various MPI application processes in a way that is both topologically-wise and performance-wise sensible. The same idea has already been used in the case of reordering: the processes that communicate a lot should be bound on two cores physically close to each other. Consequently, the sharing of caches is likely to decrease communication times and improve overall application performance.

The key idea is therefore to group processes into entities where a specific kind of resource is shared by all group[2] members (i.e. processes). MPI features a concept/construct that perfectly matches our needs: the *communicator*. As a consequence, we propose to make hardware topology information and structure available at the application level through a hierarchy of communicators. We want to help an application developer and guide him/her to build the most relevant communicator (hardware) topologically-wise, without any deep knowledge of the underlying architecture and regardless of the way the application processes are mapped and/or bound on the machine.

In this hierarchy, each communicator corresponds to a specific resource that is shared by all the processes belonging to it. For instance, if a process shares a L2 cache and a L3 cache with other processes, it will be part of the communicator encompassing all processes sharing the same L2 and part of the communicator encompassing all the processes sharing this level 3 cache. Creating communicators also allows the use of collective communications among processes that share a resource. Collective communication operations are a major feature of MPI. It gives the programmer some ability to structure his/her application and encourages him/her to improve the locality factor of the communications.

3.2 Communicators creation

There exists a couple of functions in the MPI standard which create communicators:

- `MPI_Comm_create` is able to create such an object from an MPI Group.
- `MPI_Comm_dup` duplicates a communicator taken as input argument of the function.
- `MPI_Comm_split` is able to create *subcommunicators* from the original one taken as an argument. This function yields k non-overlapping subcommunicators and the partition of the original communicator is determined by the `color` parameter (which can take `MPI_UNDEFINED` as a licit value). Processes that call this function with the same value for `color` will belong to the same output (sub-)communicator.

Since the idea behind our proposal is to create communicators based on the sharing of common resources, splitting some input communicator (`MPI_COMM_WORLD` being obviously a relevent but

[2] In the generic sense, we do not deal with the concept of MPI groups here.

not mandatory candidate) is a natural fit for the goal we want to achieve: indeed the information about the sharing of the resource can be conveyed by the `color` argument.

However, the outcome is not likely to be the one expected: let us take for example the case of processes mapped onto different physical nodes (for the sake of simplicity, each core of each node executes its own process). Let us then assume that each node features several L3 caches. If we want to create as many communicators as the number of L3 caches in our configuration, we cannot provide a single color value, otherwise, *all* processes would end up belonging to the same communicator. Ideally, we would like to provide the same color value, but this value has to carry a different meaning in different processes.

3.2.1 `MPI_Comm_split_type` extension

Fortunately, this is exactly the behaviour of the `MPI_Comm_split_type` function.
`MPI_Comm_split_type` is part of the MPI-3 shared memory programming functionnalities and features the following prototype[3]:

```
int  MPI_Comm_split_type(MPI_Comm oldcomm,
                         int split_type,
                         int key,
                         MPI_Info info,
                         MPI_Comm *newcomm)
```

With :

IN `oldcomm`: communicator (handle)

IN `split_type`: type of processes to be grouped together (integer)

IN `key`: control of rank assignment (integer)

IN `info`: info argument (handle)

OUT `newcomm`: new communicator (handle)

This function partitions the group associated with `comm` into disjoint subgroups, based on the type specified by the value assigned to the `split_type` parameter. A single value is currently defined in MPI 3.1: `MPI_COMM_TYPE_SHARED`. when this value is used, the input communicator is split into communicators, where each new communicator represents a shared-memory domain. That is, two processes belonging to the same subcommunicator are able to create a mutualy accessible shared-memory region. Obviously, there is no overlap between these new communicators. This function, along with the particular `MPI_COMM_TYPE_SHARED` value for the `split_type` parameter, already allows the user to better understand the way the processes are mapped onto the underlying hardware. It also gives the opportunity to take advantage of it since a different programming model (MPI-3 Shared Memory style) can be used in each new communicator and classical Message Passing between them.

The MPI standard stipulates that implementations may define their own values for the `split_type` parameter, in order to enforce specific behaviours. The flexibility granted by this approach is counterbalanced by its sheer lack of portability.

As a consequence, we propose to enrich the set of possible values for the `split_type` argument by adding a new one (`MPI_COMM_TYPE_PHYSICAL_TOPOLOGY` for instance[4]). The `info` argument can be used to pass hints to the implementation about the way the split should be done.

[3] Only the C version is shown.

[4] Or any suitable and meaningful name.

3.2.2 Proprieties of the hierarchical communicators

A call to `MPI_Comm_split_type` with this new value shall yield a communicator corresponding to the *highest possible level* in the hierarchy tree of the hardware topology. This newly produced communicator can then be used as an input argument in subsequent calls to `MPI_Comm_split_type` to produce other children communicators that correspond to deeper levels (see the example below for a practical use). Also, the newly produced communicators should retain the following proprieties:

- The last valid communicator produced in this fashion may be identical to `MPI_COMM_SELF`, but not necessarily.
- Each recursively created new communicator should be a strict subset of the input communicator. That is, a call to `MPI_Comm_compare(oldcomm,newcomm)` should return `MPI_UNEQUAL`. This propriety ensures that we do not create unnecessary new communicators in case of redundancies of levels in the hardware topology. For instance, if a L3 cache and a L2 cache is shared between all processes, there is no need to create a communicator for both resources.
- If no valid communicator is to be created, `MPI_COMM_NULL` should be returned.

These communicators calls will form a kind of hierarchy, mimicking the hardware one, as all new communicators are encompassed (so to speak) in their parent communicator. That is, if a process belongs to the communicator corresponding to the n-th level of the hierarchy, it also belongs to all communicators corresponding to levels 0 to $n-1$. It is to be noted that our abstraction does not make any distinction between the network and the nodes internal memory hierarchy. We give some means to organize an application according to the *structure* of the hardware, not its *nature*.

3.2.3 Creation of Roots Communicators

One other usefull addition would be the ability to create at the qame time at each level of the hierarchy another communicator which includes all root processes of a hierarchical communicator. This forms another kind of hierarchy of its own and could ease the communication between all the levels of the hierarchy. This function could have the following prototype:

```
int MPI_Comm_hsplit_with_roots(MPI_Comm oldcomm,
                               MPI_Info info,
                               MPI_Comm *newcomm,
                               MPI_Comm *rootscomm)
```

With:

IN `oldcomm`: communicator (handle)

IN `info`:info argument (handle)

IN `newcomm`: communicator (handle)

OUT `rootscomm`: communicator (handle)

`newcomm` is the same communicator created by a call to `MPI_Comm_split_type` with `MPI_COMM_TYPE_PHYSICAL_TOPOLOGY` as value for the `split_type` parameter. `rootscomm` is the communicator containing all processes that are roots in `newcomm`. A valid roots communicator can only be returned if the root process of `oldcomm` calls this function. MPI_COMM_NULL is otherwise returned by non-root processes.

3.2.4 Interaction with process mapping/binding policies

A special care should be taken regarding the current binding of process ranks. Indeed, the deepest level that shall be returned should correspond to the current process binding (e.g if a rank is bound to a L3 cache, no information below this level should be returned since it may use different L2 caches below when moving inside the binding. Any attempt to do so should return `MPI_COMM_NULL`. Moreover, unbound processes may move across an entire shared-memory machine and therefore cannot belong to a communicator split deeper than the "machine" level: in such a case, the returned communicator shall be the same as the one returned by a call to `MPI_Comm_split_type` with the `MPI_COMM_TYPE_SHARED` value for the `split_type` parameter.

3.2.5 A practical example

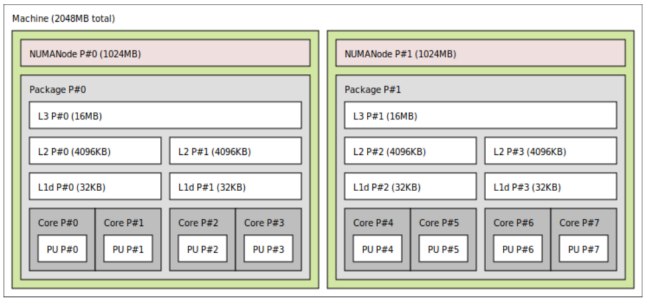


Figure 1: A hierarchical node example.

We now detail a practical example of use of this new `split_type` value. Let us suppose that an MPI application is launched on several machines featuring the memory hierarchy depicted by Figure 1: each node is composed of two NumNode with a single socket (package) and 4 cores per socket. Each socket features its dedicated L3 cache and a L2 is shared between a pair of cores. At some point, we assume that the various processes of the application execute the following code:

```
MPI_Comm newcomm[NLEVELS];
MPI_Comm oldcomm = MPI_COMM_WORLD;
int rank, idx = 0;
while(oldcomm != MPI_COMM_NULL){
  MPI_Comm_rank(oldcomm,&rank);
  MPI_Comm_split_type(oldcom,
                      MPI_COMM_TYPE_PHYSICAL_TOPOLOGY,
                      rank,
                      MPI_INFO_NULL,
                      &newcomm[idx]);
  oldcomm = newcomm[idx++];
}
```

In this code snippet, `NLEVELS` is chosen appropriety so that there are enough elements in the `newcomm` array, but as we shall discuss later (see Section 3.4), our proposal does not make any assumption on the total number of levels in the hardware hierarchy nor on the nature of the hardware resource the communicator is supposed to represent for the processes.

A simple case In our first case, we suppose that each process is bound to its own core. More precisely, process rank p_i (in `MPI_COMM_WORLD`) is bound to core number i. When the code shown previously is executed, the following communicators are created:

- `newcomm[0]` $= \{p_0, p_1, p_2, p_3, p_4, p_5, p_6, p_7\}$
- `newcomm[1]` $= \{p_0, p_1, p_2, p_3\}$ (for NumaNode 0) and `newcomm[1]` $= \{p_4, p_5, p_6, p_7\}$ (for NumaNode 1)
- `newcomm[2]` $= \{p_{2i}, p_{2i+1})\}$ (for each L2 number i) : 4 communicators
- `newcomm[3]` $= \{p_i\}$ (for each Core number i) : 8 communicators

As for the roots communicators we shall have the following situation (we suppose that `rootscomm[i]` corresponds to the hierarchy level `newcomm[i]`):

- `rootscomm[0]` $= \{p_0\}$
- `rootscomm[1]` $= \{p_0, p_4\}$
- `rootscomm[2]` $= \{p_0, p_2\}$ and $\{p_4, p_6\}$
- `rootscomm[3]` $= \{p_0, p_1\}$, $\{p_2, p_3\}$, $\{p_4, p_5\}$ and $\{p_6, p_7\}$

A slightly more complicated case Let us now suppose that the processes are not bound to a specific core: some processes are bound to cores, some others are bound on the L2 cache and some on a NumaNode. Figure 2 depicts such a case involving 8 processes. P0 and P1 are bound

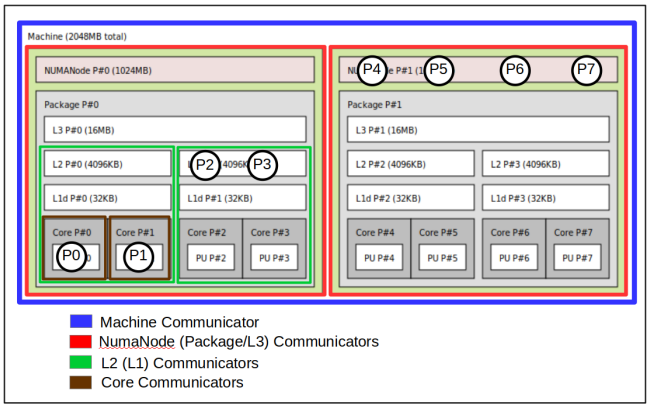


Figure 2: A case of non-uniform binding policy for processes

to their own cores, P2 and P3 are bound to a L2 cache, while P4, P5, P6 and P7 are bound to a NumaNode/Package/L3 cache. In this case, and according to the proprieties of the hierarchical communicators:

- `newcomm[0]` $= \{p_0, p_1, p_2, p_3, p_4, p_5, p_6, p_7\}$
- `newcomm[1]` $= \{p_0, p_1, p_2, p_3\}$ (for NumaNode 0) and `newcomm[1]` $= \{p_4, p_5, p_6, p_7\}$ (for NumaNode 1)

- `newcomm`[2] = $\{p_0, p_1\}$ (for L2 0) and
  `newcomm`[2] = $\{p_2, p_3\}$ (for L2 1)
- `newcomm`[3] = $\{p_0\}$ (for Core 0) and
  `newcomm`[3] = $\{p_1\}$ (for Core 1)

The roots communicators would then be:

- `rootscomm`[0] = $\{p_0\}$
- `rootscomm`[1] = $\{p_0, p_4\}$
- `rootscomm`[2] = $\{p_0, p_2\}$
- `rootscomm`[3] = $\{p_0, p_1\}$

This demonstrates that our proposal is flexible enough to accommodate the case of non-uniform binding policies within the same MPI application.

3.3 Communicators characteristics query

The main aspect of our proposal deals with hierarchical communicators creation. However, we need to introduce other functions in order to make the use of these communicators more practical to the application developer. So far, with the proposed abstraction, users are able to leverage the structure of their hardware. However, more information might be needed, for instance in case of data distribution between the various communicators created at a certain level in the hierarchy.

3.3.1 Getting information for a hierarchical level

A process is able to retrieve some information about a specific communicator it belongs to with a call to the follwing function:

```
int MPI_Comm_get_hlevel_info(MPI_Comm comm,
                             int *num_comms,
                             int *index,
                             char **type)
```

With:

IN `comm`: communicator (handle)

OUT `num_comms`: number of siblings communicators (integer)

OUT `index`: communicator index (integer)

OUT `type`: type of communicator (string)

- `num_comms` is the number of communicators in the same level with the same parent communicator. For instance, in the case of a node similar to the one shown on Figure 1, a call to `MPI_Comm_get_`
  `hlevel_info` with `newcomm[2]` (that is, a L2 cache) would yield a value of 2 for `num_comms`, since there are 2 L2 caches per NumaNode (i.e. `newcomm[1]`, the parent communicator).
- `index` is a kind of rank for each communicator which should be contiguously numbered and starting from 0. It is the rank of the communicator among all communicators created by its parent communicator.

- `type` is a string giving information about the kind of resource that the communicator represents. It should be unambiguous, like `L2_Cache`, `L3_Cache` or `NumaNode`.

All this information should be cached by the communicator in an info object attached to it containing a set of *(key,value)* proprely defined. This would require the use of the `MPI_Comm_set_info` and `MPI_Comm_get_info` functions.

3.3.2 Getting the minimal level

Another helpful feature would be the ability for a programmer to know the name (type) of the *lowest level* in the hardware hierarchy that is shared by some processes. To this end, we propose to add the following function:

```
int MPI_Comm_get_min_hlevel(MPI_Comm comm,
                            int nranks,
                            int *ranks,
                            char **type)
```

With:

IN `comm`: communicator (handle)

IN `nranks`: number of MPI processes (integer)

IN `ranks`: list of MPI process ranks (array)

OUT `type`: type of the resource (string)

This function returns the name of the *lowest* level in the hierarchy shared by all the MPI processes which ranks in the communicator `comm` are listed in the `rank` array. If the calling process rank is not among the ranks listed in the array passed as an argument, the type returned should be "`Unknown`" or "`Invalid`".

3.4 Discussion

In this section, we discuss the design choices, advantages and potential drawbacks of our approach. We propose an abstraction based on existing MPI objects (the communicators), hierarchically modelling the hardware topology in order to improve performance and scalability. Communicators are often used in MPI applications and a well-understood concept to boot. A large majority of application developers are familiar with it (beyond `MPI_COMM_WORLD`). Therefore, using our proposal would require little effort, conceptually speaking. The communicators created do not feature a predetermined name, taken after the underlying resource it is supposed to represent. This ensures that there will no need to make any change nor modifications in the future in case new levels in the hardware hierarchy should appear. This also justifies why we do not specify a maximal depth for the hierarchy. By creating "recursively" new communicators, the user is able to get all needed objects until the bottom is reached and no new communicator can be produced. The user can of course chose the desirable depth by querying the name/type of the communicator created and deciding to go further or not. This choice also explains why we do not rely on several predetermined values for the `split_type` argument [5] (e.g. one value corresponding to a specific hierarchy level) because architectural changes in hardware would require modifications to the standard.

[5] As implemented in Open MPI with the `OMPI_COMM_TYPE_*` values for instance.

Currently, our design is based on a “recursive" call to `MPI_Comm_split_type` in order to create the hierarchy. However, an alternate solution would be to introduce a function that create all communicators at once, returning an array for instance as well as its size (i.e. the hierarchy depth). Such a function is more simple to use, but compells the user to create all the levels, even undesired ones. The chosen design allows for more control at the cost of some ease of use.

One drawback of our approach is the fact that it targets architectures which are hierarchically organized. It is the case for most machines, but there are some exceptions that our model does not currently address. And the same issue arises for network topologies that are not hierarchical, such as torus for instance. In such a case, should the split be made on just a particular dimension of the torus? The `info` argument could be used to pass this information. But if some form of hierarchy can be extracted from the network topology, we shall be able to exploit it with our mechanism.

Currently, the interface proposed features one new value for the `split_type` argument of the `MPI_Comm_split_type` function and three new routines. But one open question is whether or not we should offer more elements. Indeed, it could be interesting to create several hierarchies of communicators, based on different criteria with different `split_type` values. For instance, making a distinction between the network topology and the nodes internal topology could be a relevant idea. The info argument in `MPI_Comm_split_type` could also be used to split a communicator directly at a desired level, without creating the whole hierarchy.

As for the names (types) returned by the `MPI_Comm_get_hlevel_info` function, they could be given or derived from the names used by an external tools such as hwloc. By doing so, portability could be enforced to some degree, or at least across MPI implementations that feature hwloc (which is the case of Open MPI, MPICH and MVAPICH for instance).

Last, its is possible to create the roots communicators with the current functions available in MPI. However, from a performance standpoint, an hwloc-based implementation (and thus a new function in the standard) is more efficient. It also removes the burden of implementing it by the user.

4 Benefits of the hierarchical model for parallel applications

In this section we propose to model the class of MPI applications that can leverage our hierarchical abstraction to enhance the performance of their communications. Let us suppose that an

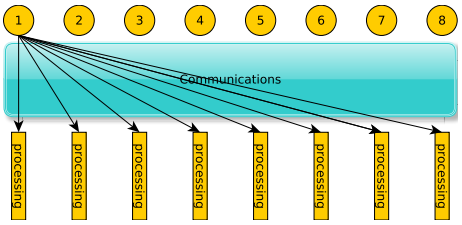


Figure 3: Superstep of MPI application familly based on native collective communications

MPI application features p MPI processes. The application execution model we focus on is in the form of a repetitive phases of super-steps including communication operations C followed by execution operations E as shown by Figure 3. Each process needs to terminate its communication operation to start the processing of data and its pseudo code is:

If the communication operations C include collective operations, then the application can take advantage of the hierarchical model described by Algorithm 2, where L is the number of levels in

Algorithm 1: MPI algorithm covered by our hierarchical model

```
1 MPI_Initialization(); (// Initialisation part) foreach s← S − 1..0 do
    // Algorithm super-steps
2   MPI_Collective_Communication_Operations(); MPI_Processing_Operations();
```

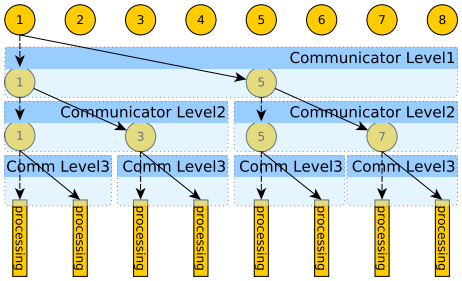


Figure 4: Superstep of MPI application familly based on hierarchical collective communications

the hierarchy allowed by the target architecture and generated automatically by our approach. Figure 4 illustrates the application communication model based on a three-level communicator hierarchy.

Algorithm 2: MPI algorithm based on a hierarchical communication model

```
1 MPI_Initialization();(// Initialisation part) while L ≠ MPI_Comm_NULL do
2   L ← MPI_Comm_hsplit_with_roots(); // Topological Communicators
      creation
3 foreach s← S − 1..0 do // Algorithm super-steps
4   foreach l← L − 1..0 do // Hierarchy levels
5     MPI_Collective_Communication_Operations(l);
6   MPI_Processing_Operations();
```

Applying a hierarchical model can improve the communication times and thus the total time of the application. As a matter of fact, by exploiting more parallelism in the hierarchy levels, it is possible to enhance the scalability and performance of collective operations. In order to illustrate this claim we propose a short discussion to highlight the important features which can explain this benefit.

The collective communications are modeled with well-known works such as the Hockney [6], the LogP [3] or the PlogP [11] model. These models express the maximum time taken by a collective communication operation as a linear f unction depending on several variables such as:

- p: The number of processes performing the collective communication operation c.
- m: The size of the data exchanged between processes.
- The collective algorithm used.
- α: The hardware latency that may be a function of m and p.
- β: The hardware bandwidth that may be a function of m and p.

For example, the processing time of the simple Flat Tree broadcast is expressed with the Hockney [6] model as equal to :

$$(p-1) \times (\alpha + m\beta)$$

As shown by Algorithm 2 L levels of communications are present in the topology-aware hierarchical model according to the structure of the architecture. Thus, the time of collective operations will be represented as a sum of L functions depending on the previous cited features but where each level is characterised by a set of specific parameters: p_i, α_i, β_i and m_i. As a consequence, the Flat Tree broadcast example shall be expressed as:

$$\sum_{i=0}^{L-1}(p_i - 1) \times (\alpha_i + m\beta_i)$$

The hardawre topology-aware hierarchical model could reduce the processing time of collective operations by exploiting the following points:

- Less number of processes per level: in most applications, it is enough to use a pyramidal structure in the hierarchical model of computation. In fact, as illustrated by Figure 4, the structure based only on the roots of hierarchical communicators in upper levels is enough to perform the communication. such a structure implies to use fewer processes p_i by level, which reduces the processing time of the collective.
- Parallelism by level: this point regards the exploitation of parallelism at each level of the hierarchy. Indeed, and except for the top level of the hierarchy, the communications inside the communicators of the same level are carried out in parallel.
- Data locality and process affinity: This point highlights the advantage of using hardware-aware communicators. Indeed, taking into account the hardware affinity of processes placement in the machine, the communication between them is enhanced thanks to the cache optimizations.
- Improved latency and bandwidth per level: this point concerns the physical latency and bandwidth which could be improved when the number of processes is small. In fact, because of the contention phenomenon, the higher the number of processes involved in the communication at the same time through an interconnect, the higher the latency and the lower the bandwidth delivered to each process.

From all the above points, collective communication operations could be improved depending on the execution conditions. For instance, if we take the Flat Tree broadcast of a single message ($m = 1$) and operated in the simple context of 8 processes and the three-level hierarchy as described by Figure 4 then the result is:

$$\text{The simple case: } Time = (8-1)(\alpha + \beta)$$

$$\text{The hierarchical case: } Time = (2-1)(\alpha_0 + \beta_0) + (2-1)(\alpha_1 + \beta_1) + (2-1)(\alpha_0 + \beta_1)$$

If we compare both expressions, it is clear that the hierarchical approach enhances the simple collective with the factor of at least (7/3) times when latencies and bandwidths are equal in both the simple and the hierarchical cases. However, in the real situation, the simple execution could generate more contention on the interconnects of each level than the hierarchical execution. Thus, the factor of enhancement of collective time could be greater than (7/3).

5 Experimental Results

In this section we present the experimentations we carried out their results to demonstrate the benefits of our proposal. We modified two benchmarks by introducing our hierarchical model of communicators and we compared their executions on three architectures. In all cases we used the roots communicators hierarchy generated by calling the primitive described in subsection 3.2.3. In addition, we used all cores of a targeted architecture and bound an MPI process on each of them.

5.1 Platforms and architectures

For our study, we used tree architectures: a network of 10 nodes (NTW10E5) and two SMP machines (SMPE12E5, SMP20E5) from the Plafrim platform [17]. The characteristics of these architectures are given in Table 1.

Table 1: Characteristics of the used architectures

Name	SMP12E5	SMP20E7	NTW10E5
OS	Red Hat 4.8	SUSE Srv 11	Red Hat 4.8
Kernel	3.10.0	2.6.32.46	3.10.0
Nodes	1	1	10
Cores per NUMA	8	8	6
NUMA nodes	12	20	4
Sockets	12	20	2
NUMA groups	12	20	1
Socket	E5-4620	E7-8837	E5-4620
Clock rate	2600Mhz	2660Mhz	2600Mhz
Hyper-Threading	Yes	No	No
L1 cache	32K	32K	32K
L2 cache	256K	32K	256K
L3 cache	20480K	24576K	15360K
Mem Interconnect	NUMAlink6	NUMAlink5	QPI
Node Interconnect	N/A	N/A	InfiniBand
Hierarchical levels	3	3	4
GCC	5.1	5.1	5.1
OpenMPI	2.0.1	2.0.1	2.0.1
Hwloc	2.0-git	2.0-git	2.0-git

5.2 Collective Communications

The first benchmark is the broadcast and the reduce collective communication operations which we chose to test the proposed hierarchical communication model. There has been other works that aimed at exploiting hierarchy in the hardware in order to improve collective communications. In [23] a distinction is made between inter-node and intra-node communication, because shared-memory based communications are expected to be faster than their network-based counterparts. In this case, the hierarchy was limited to two levels (intra vs. inter-node) and sometimes three (intra. vs inter-cluster). We generalized this approach and make no assumption about the number of levels. Moreover we are able to exploit the memory hierarchy inside the nodes of a cluster which is not addressed at all by these works. However, the aim here is *not* to rewrite

collectives or to propose some new algorithms. Our goal is merely to experiment our abstraction in order to assess the potential gains achievable by optimizing communication and data locality.

We compared two implementations:

- *Native*: this is the Open MPI implementation of the considered collective.
- *Hierarchical*: this is a loop over the levels of the hierarchy calling the Open MPI version of the collective. Hence, we do not rewrite the collective but just call it through our hierarchy.

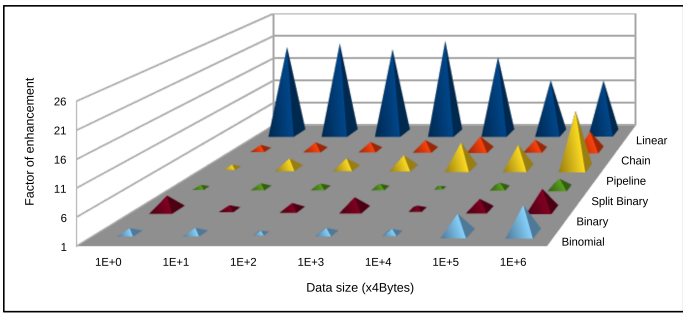


Figure 5: Enhancement factor of hierarchical approach for Open MPI Broadcast implementations on NTW10E5 (240cores)

Figures 5, 6 and 7 show the enhancement factor of the hierarchical-based implementation of six broadcast algorithms: Linear, Chain, Pipeline, Split Binary, Binary and Binomial. This factor is obtained by comparing the maximum time of processing several data sizes on the architectures described in Table 1 with the hierarchical broadcast and the native version of Open MPI broadcast. It is possible to note that the hierarchical approach we propose enhances almost all broadcast executions on the three architectures. In fact, the maxima achieved are roughly equal to 21x on NTW10E5, 11x on SMP20E5 and 22x on SMP12E5. This performance gain is due to two majors factors: first, our hierarchical approach allows to better exploit the parallelism and to reduce the complexity of broadcast algorithms. Second, the hardware topology-based communicators enhance the data locality and the hardware affinity between processes performing communications. Third, the communications are better pipelined over the network and the interconnect which reduce their total time.

Figures 8, 9 and 10 present the same results for the Reduce collective with six algorithms: Linear, Chain, Pipeline, Binary, Binomial and In-order Binary. In this case also the enhancement factor is obtained by comparing the maximum time for executing the reduction collective of several data sizes on the architectures described in Table 1 with both hierarchical and native Open MPI reduce. Here, we note that the hierarchical approach we propose considerably enhances the first three algorithms: Linear, Chain, Pipeline. In fact, the maxima achieved are roughly equal to 39x on NTW10E5, 11x on SMP20E5 and 8x on SMP12E5. These performances are achieved by the hardware-aware and hierarchical decomposition of the communications. Indeed, the algorithms are better parallelized, the data-locality is enhanced and the communications are better pipelined. However, we note that the three last algorithms (e.g. Binary, Binomial and In-order Binary) are less improved than the others if not at all. This phenomenon is due to the structure of the algorithms since they are already based on a hierarchical, tree-based structure. Therefore, the algorithm complexity is not enhanced by our approach. The small performances obtained for these algorithms are only due to the hardware optimisations and could be more significant with a larger number of processes.

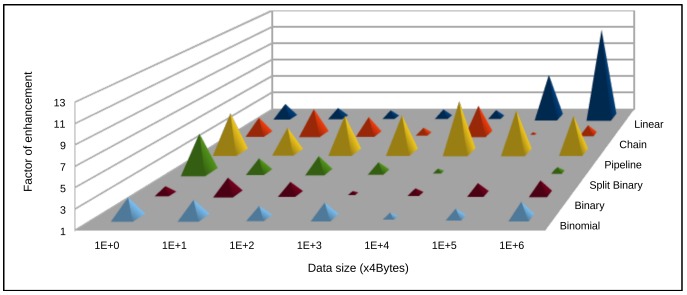


Figure 6: Enhancement factor of hierarchical approach for Open MPI Broadcast implementations SMP20E7 (160 cores)

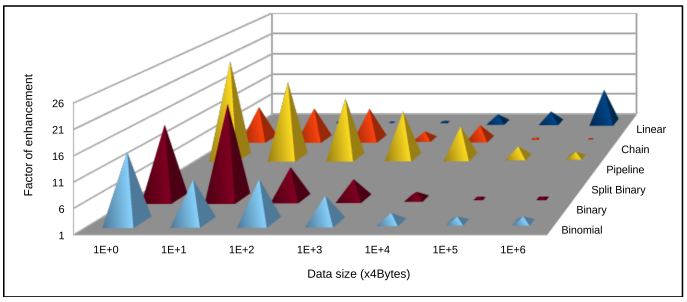


Figure 7: Enhancement factor of hierarchical approach for Open MPI Broadcast implementations SMP12E5 (96 cores)

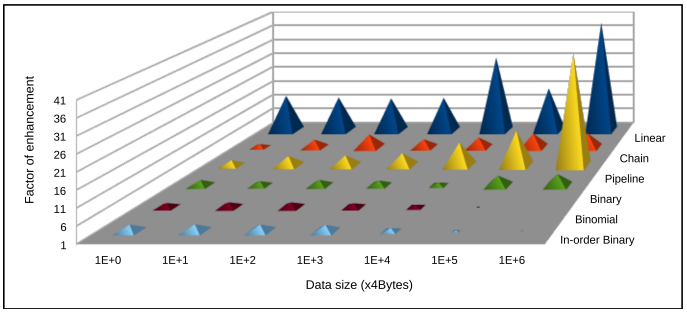


Figure 8: Enhancement factor of hierarchical approach for Open MPI Reduce implementations on NTW10E5 (240cores)

5.3 Hierarchical Matrix Multiplication

The second benchmark we carried out is a two-dimensional, hierarchical matrix multiplication [18]. This application is an hierarchical extension of the SUMMA [21] algorithm based on two levels of hierarchy. Its implementation is hardware oblivious and is based on exploring several hierarchy configurations i.e. different number of communicators on each two levels. Hence, the programmer needs to manually specify the configuration at each execution by giving a configuration file as an input argument. With our approach, we abstract the manual specification of the hierarchy configuration. As a consequence we do not need to specify the configuration of

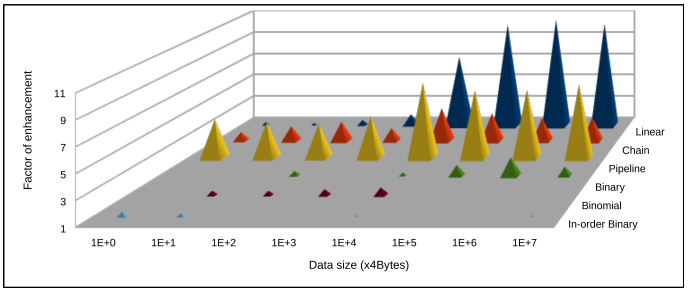


Figure 9: Enhancement factor of hierarchical approach for Open MPI Reduce implementations on SMP20E7 (160 cores)

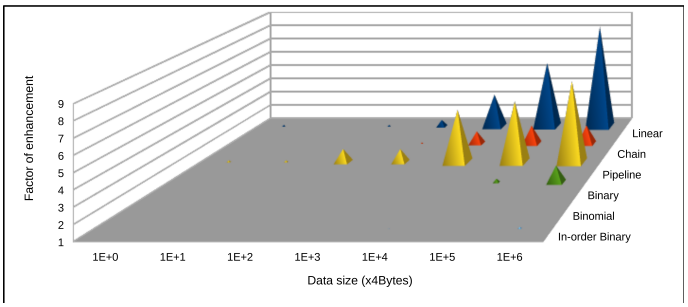


Figure 10: Enhancement factor of hierarchical approach for Open MPI Reduce implementations on SMP12E5 (96 cores)

the hierarchy in our implementation. It is based on the targeted architecture topology and automatically generated by using our proposed set of functions. The implementations we compare are:

- *Simple*: this is the implementation of the SUMMA algorithm based on broadcasting the blocks over rows and columns.
- *Hierarchical (xgroups)*: this is the hierarchical implementation of the SUMMA algorithm of matrix multiplication. The broadcasts of blocks are performed hierarchically over two levels. The hierarchy configuration is set by the user through a configuration file for each execution.
- *Hierarchical topological*: this is the same implementation but leveraging our communicators hierarchy. The hierarchy configuration is based on the underlying hardware topology and automatically build using our primitives.

Figures 11, 12 and 13 present the total processing times of several implementations: Simple (without hierarchy), Hierarchical with various configurations and Hierarchical with a topological configuration. The total time represents the needed time to process one block of 8 doubles per process and is composed of the average computing time represented by the blue part of the bar and the average communication time of processes by the red part. The used broadcast algorithm is the Open MPI implementation of Binary algorithm. We note that the Topological implementation (last bar) represents the minimum total time and achieve the better speed-ups of the simple implementation: 12x on NTW10E5, 5x on SMP20E7 and 6x on SMP12E5. Indeed,

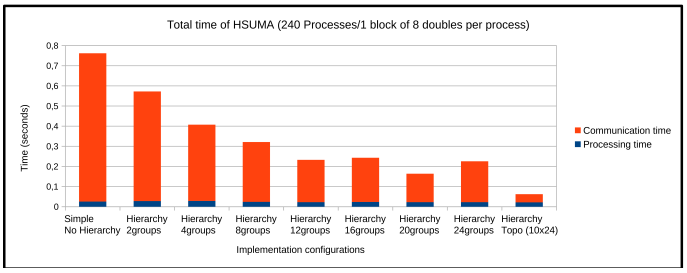


Figure 11: Comparison of execution time of simple, topology-oblivious hierarchical and toplogical implementations on NTW10E5. 240 processes process 1 block of 8 doubles

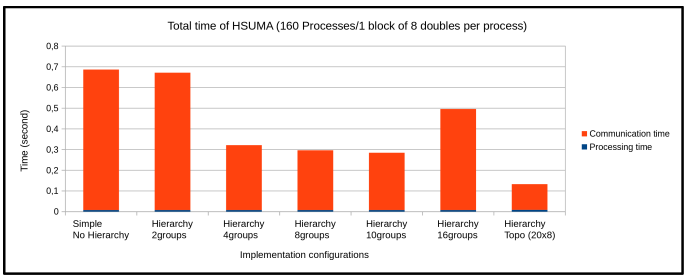


Figure 12: Comparison of execution time of simple, topology-oblivious hierarchical and toplogical implementations on NTW10E5. 160 processes process 1 block of 8 doubles

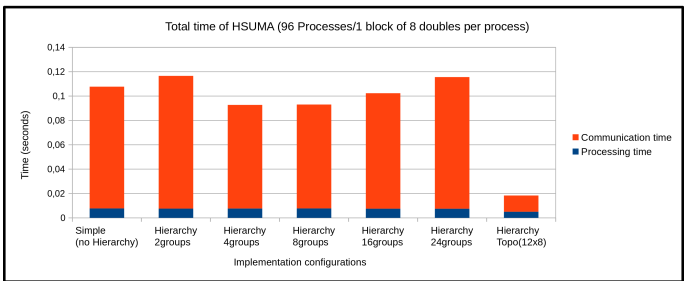


Figure 13: Comparison of execution time of simple, topology-oblivious hierarchical and toplogical implementations on NTW10E5. 96 processes process 1 block of 8 doubles

all implementations feature the same processing time (the blue part) but the communication time (the red part) is optimally reduced by the last implementation (Topological). This is due to the optimal hardware configuration matching with the topology. In fact, the communications inside hardware topology-aware communicators are more efficient thanks to the data locality and the MPI processes affinity. In addition, pipelining the communication enhances processors occupation time and reduces the global time.

6 Conclusion and Future Work

In this paper, we have presented an abstraction that can help the programmer to structure his/her application in order to take into account the hardware topology while he is designing it. We also presented how this model and abstraction could fit into an existing programming model/standard: the widely-used Message Passing Interface. At the expense of light changes and a few new features introduction, we have shown that performance improvements can be achieved in MPI implementations themselves, but more importantly in parallel applications directly. Indeed, we introduced our topology-based communicators in two collective communication operations (Broadcast and Reduce) and showed that in the cases where hierarchy could improve performance, taking into account the physical topology improves things even further. Tested on three different architectures, our approach enhances the performance of the tested collective communications by factors up to 21x, 11x and 22x for the broadcast and 39x, 11x and 8x for the reduce. It also reduces the total time of hierarchical matrix multiplication with the factors of 12x, 5x and 6x. Even if these gains of performance are considerable, the additional effort to use our approach is negligible. In fact, thanks to the proposed abstraction automatically generating an hierarchy of communicators, the user does not have to deal with the details of hardware characteristics or manipulating low-level tools. He only deals with high-level MPI objects: communicators.

The hardware-agnosticity nature of MPI might seem paradoxical with our goal, as we precisely seek to offer the programmer means to better understand and exploit the underlying hardware. We believe that the independence from hardware considerations is an important strength of the MPI standard. We intend to keep MPI hardware-agnostic but we also believe that giving the programmer more hints about this same hardware can be very beneficial performance-wise. The predicament is therefore to find the relevant level of abstraction for such a new functionality. Indeed, if an MPI application should be able to gather and use specific pieces of information about the hardware, this information should nevertheless be abstract enough to not be tailored for a particular class of hardware. This work is available as an external library that features all the functions presented in this paper. The code is downloadable at the following URL: `https://gforge.inria.fr/frs/download.php/file/36832/hsplit-rc-0.1.tar.gz` or via a git clone from URL: `https://scm.gforge.inria.fr/anonscm/git/mpi-topology/mpi-topology.git` We plan to further discuss this proposal at the MPI Forum. Our future works include the management of networks in our implementation. Currently, we only address a node internal topology (i.e. the memory hierarchy) and we do not create communicators corresponding the the various switch levels (for instance) that are present. This is only a matter of implementation as the concept we propose is ready yet for this aspect. As we use hwloc for memory hierarchies, we would like to support networks by integrating the Netloc [4] software in our implementation. We also plan to address non-hierarchical topologies, especially regarding the network. Expressing the topology with a distance function seems a promising idea and we would like to explore it. Last, we believe that other objects in MPI implementations besides communicators could benefit from a knowledge of the underlying physical topology, for instance memory windows. Generalizing our approach could be of interest.

Acknwoledgements

This work has partially been supported by the PIA ELCI project of the French FSN and by the ANR MOEBUS project ANR-13-INFR-0001. Experiments presented in this paper were carried out using the PLAFRIM experimental testbed, being developed under the Inria PlaFRIM development action with support from Bordeaux INP, LABRI and IMB and other entities: Conseil

Régional d'Aquitaine, Université de Bordeaux, CNRS and ANR in accordance to the programme d'Investissements d'Avenir (see `https://www.plafrim.fr/`). The authors would like to thank the MPI Forum for its feedback, especially Daniel Holmes.

References

[1] B. Brandfass, T. Alrutz, and T. Gerhold. Rank Reordering for MPI Communication Optimization. *Computer & Fluids*, January 2012.

[2] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. Hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In *Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP2010)*, Pisa, Italia, February 2010. IEEE Computer Society Press.

[3] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: Towards a realistic model of parallel computation. *SIGPLAN Not.*, 28(7):1–12, July 1993.

[4] Brice Goglin, Joshua Hursey, and Jeffrey M. Squyres. Netloc: Towards a comprehensive view of the HPC system topology. In *43rd International Conference on Parallel Processing Workshops, ICPPW 2014, Minneapolis, MN, USA, September 9-12, 2014*, pages 216–225. IEEE Computer Society, 2014.

[5] T. Hatazaki. Rank Reordering Strategy for MPI Topology Creation Functions. In V. Alexandrov and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 1497 of *Lecture Notes in Computer Science*, pages 188–195. Springer Berlin / Heidelberg, 1998. 10.1007/BFb0056575.

[6] Roger W. Hockney. The communication challenge for mpp: Intel paragon and meiko cs-2. *Parallel Comput.*, 20(3):389–398, March 1994.

[7] J. Hursey, J. M. Squyres, and T. Dontje. Locality-Aware Parallel Process Mapping for Multi-core HPC Systems. In *2011 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 527–531. IEEE, 2011.

[8] J. L. Träff. Implementing the MPI Process Topology Mechanism. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–14, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[9] Emmanuel Jeannot, Guillaume Mercier, and François Tessier. Process Placement in Multicore Clusters: Algorithmic Issues and Practical Techniques. *IEEE Trans. Parallel Distrib. Syst.*, 25(4):993–1002, 2014.

[10] Jesper Larsson Träff. Implementing the MPI process topology mechanism. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–14, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[11] Thilo Kielmann, Henri E. Bal, and Kees Verstoep. *Fast Measurement of LogP Parameters for Message Passing Platforms*, pages 1176–1183. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

[12] Andi Kleen. A NUMA API for Linux. *Novel Inc*, 2005.

[13] G. Mercier and J. Clet-Ortega. Towards an Efficient Process Placement Policy for MPI Applications in Multicore Environments. In *EuroPVM/MPI*, volume 5759 of *Lecture Notes in Computer Science*, pages 104–115, Espoo, Finland, September 2009. Springer.

[14] G. Mercier and E. Jeannot. Improving MPI Applications Performance on Multicore Clusters with Rank Reordering. In *EuroMPI*, volume 6960 of *Lecture Notes in Computer Science*, pages 39–49, Santorini, Greece, September 2011. Springer.

[15] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 3.0. Technical report, September 2012.

[16] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996.

[17] Plafrim. Plate-forme fédérative pour la recherche en informatique et mathématiques. `https://plafrim.bordeaux.inria.fr/doku.php`.

[18] Jean-Noel Quintin, Khalid Hasanov, and A. Lastovetsky. Hierarchical parallel matrix multiplication on large-scale distributed memory platforms. In *42nd International Conference on Parallel Processing (ICPP 2013)*, pages 754–762, Lyon, France, 1-4 October 2013. IEEE, IEEE.

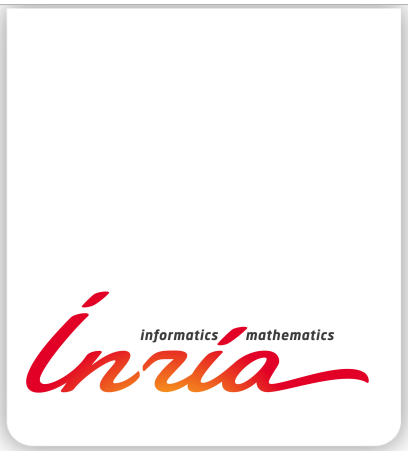
[19] M. J. Rashti, J. Green, P. Balaji, A. Afsahi, and W. Gropp. Multi-core and Network Aware MPI Topology Functions. In Y. Cotronis, A. Danalis, D. S. Nikolopoulos, and J. Dongarra, editors, *EuroMPI 2011. Recent Advances in the Message Passing Interface - 18th European MPI Users' Group Meeting*, volume 6960 of *Lecture Notes in Computer Science*, pages 50–60. Springer, 2011.

[20] James Reinders and Jim Jeffers. *High Performance Parallelism Pearls*, volume 2. Morgan Kaufmann, 1 edition, 2015.

[21] R. A. Van De Geijn and J. Watts. Summa: scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997.

[22] J. Zhang, J. Zhai, W. Chen, and W. Zheng. Process Mapping for MPI Collective Communications. In H. J. Sips, D. H. J. Epema, and H.-X. Lin, editors, *Euro-Par*, volume 5704 of *Lecture Notes in Computer Science*, pages 81–92. Springer, 2009.

[23] H. Zhu, D. Goodell, W. Gropp, and R. Thakur. Hierarchical Collectives in MPICH2. In *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 325–326, Berlin, Heidelberg, 2009. Springer-Verlag.



RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399