Process Placement in Multicore Clusters: Algorithmic Issues and Practical Techniques

Emmanuel Jeannot, Guillaume Mercier, and François Tessier

Abstract—Current generations of NUMA node clusters feature multicore or manycore processors. Programming such architectures efficiently is a challenge because numerous hardware characteristics have to be taken into account, especially the memory hierarchy. One appealing idea to improve the performance of parallel applications is to decrease their communication costs by matching the communication pattern to the underlying hardware architecture. In this paper, we detail the algorithm and techniques proposed to achieve such a result: first, we gather both the communication pattern information and the hardware details. Then we compute a relevant reordering of the various process ranks of the application. Finally, those new ranks are used to reduce the communication costs of the application.

Index Terms—Parallel programming, high performance computing, multicore processing

1 Introduction

In the fields of science and engineering, it is necessary to solve complex problems that require a tremendous amount of computational power (e.g., molecular dynamics, climate simulation, plane wing design, etc.). Nowadays, for such applications, parallel computers are used in order to solve larger problems at longer and finer time-scales. However, with the expected increase of application concurrency and input data size, one of the most important challenges to be addressed in the forthcoming years is that of *locality*, i.e., how to improve data access and transfer within the application [1].

Among the different aspects of locality, one issue arises from the memory and the network: the transfer time of data exchanges between processes of an application depends on both the affinity of the processes and their location. A thorough analysis of the way an application behaves and of the platform on which it is executed, as well as clever algorithms and strategies have the potential to dramatically improve the application communication time. Indeed, the performance of many existing applications could benefit from improved locality [2]. In this paper, we therefore tackle this locality problem by optimizing data transfers between processes of an application. The proposed solution relies on models of the processes' affinity and on models of the topology of the underlying architecture. We use an algorithm called TreeMatch to perform an optimized process placement that tells where to map these processes on the computing units of a distributed memory multicore parallel machine.

This paper exposes the model, TreeMatch and some optimizations as well as the techniques we developed to compute and enforce a placement policy. Moreover, for validation purposes, this work has been instantiated using the message passing interface (MPI) [3]. Experiments show that our algorithm, thanks to its adaptive strategies, is able to execute faster than other usual techniques, such as graph-embedding or graph partitioning. On synthetic kernels and on a real-world computational fluid dynamics (CFD) application, we show that, by placing the processes so that the communication pattern matches the underlying hardware architecture, substantial performance gains can be achieved compared to standard MPI placement policies and other solutions from the literature. Moreover, this placement can be enforced automatically and transparently thanks to the virtual topology mechanisms available in the MPI standard [4].

The work exposed in this paper expands and enhances two prior works. Compared to the first version of TreeMatch published in [5], the new enhanced version features a complexity reduced from exponential to polynomial. Also, the study is carried out in the context of distributed memory systems with comparisons to state-of-the art solutions: MPIPP and (new in this paper) Chaco, Metis, Scotch. The integration of TreeMatch in MPI implementations, as described in [6], is centralized and therefore lacks scalability. The new version, partially distributed, addresses such scalability issue. Last, all of the experimental results shown in Section 5 are unpublished ones.

This paper is organized as follows: Section 2 exposes the problem and the method used for this work. Section 3 describes previous and related works dealing with process placement, while the core of our work, TreeMatch, is described and discussed in Section 4. Experiments that validate our approach are analyzed in Section 5, and Section 6 concludes this paper.

2 Problem Statement and Method Description

This work targets process placement to tackle the locality problem that stems from the way data are exchanged between processes of a parallel application either through the network or through the memory. De facto,

• *The authors are with the INRIA Bordeaux Sud-Ouest, 200 Avenue de la Vieille Tour Talence 33405.*
E-mail: {emmanuel.jeannot, francois.tessier}@inria.fr, mercier@labri.fr.

Manuscript received 6 Aug. 2012; revised 25 Jan. 2013; accepted 25 Mar. 2013; date of publication 7 Apr. 2013; date of current version 21 Feb. 2014.
Recommended for acceptance by M.E. Acacio.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2013.104

the main standard for programming with parallel processes is MPI. Therefore, in the remainder of this paper, the problem is tackled through the prism of MPI. Nevertheless, most of this work it is not strictly bound to this standard. It can be applied to any other parallel process-based programming model. For instance, it can be applied to Charm++ [7] and to a lesser extent to partitioned global address space (PGAS) languages.

Moreover, we would like to emphasize the fact that the method and algorithm presented in this paper make no assumptions about the MPI processes themselves. Our work is thus applicable even in the case of multithreaded MPI processes: this only requires that the cores executing the threads of a given process be considered as a single processing element. To account for this abstraction, we will refer to *computing units* in order to encompass both notions of cores and processors.

An MPI application distributes its work among entities called *MPI processes* that run in parallel on the various physical computing units of the machine (processors or cores). The programming model of MPI is *flat*: each process can communicate directly with other application processes. All processes send and receive messages containing data during the application execution. The exchanges can be irregular, which means that a given MPI process will not necessarily communicate with all the other MPI processes and that the amount of data exchanged between consecutive messages may vary. This *communication pattern* can be viewed as a characteristic of the application [8]. Several examples of such patterns can be found in Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2013.104.

On the other hand, MPI applications can run on a wide range of hardware architectures. In the case of clusters of NUMA nodes, both the network and the nodes' internal memory hierarchy induce communication speed variations. For instance, two processes sharing the same L3 cache will communicate faster than two processes located on different nodes. As a consequence, the physical location of the MPI processes influences application communication costs. That is, communication performance is heterogeneous within a single machine. An intuitive idea is therefore to match an application communication pattern to the target hardware by mapping the application processes onto dedicated computing units.

Process placement is of interest for classes of parallel applications for which performance is limited by the communication efficiency (a.k.a communication-bound applications). The current trend in parallel architectures is to increase the number of computing units as much as possible. However, what is possible with processors and cores is not with the memory resources. Hence, the amount of available memory per computing unit is likely to decrease drastically in the forthcoming years. As a consequence, the process placement issue is relevant even for compute-bound applications as in the near future the memory, and later the network, might become the bottleneck of some of them. Hence, decreasing communication costs is important to improve scalability, regardless of the class of parallel applications considered.

To compute such a mapping, we propose the following three-steps method:

1. Gather the communication pattern of the target application.
2. Model the target underlying architecture.
3. Compute a matching between the MPI process ranks and the computing unit numbers. This matching defines a *placement policy* that is enforced when the application is launched.

2.1 First Step: Gathering an Application Communication Pattern

The first piece of information needed is the target application communication pattern. Currently, our method relies on the instrumentation of the application code followed by a preliminary run of this modified code. To that end, we introduced a limited number of profiling elements[1] within existing MPI implementations (both MPICH2 and Open MPI). By modifying the low-level communication layers in the MPICH2 (e.g., the Nemesis channel [9]) and Open MPI stacks, we are able to trace data exchanges exhaustively in cases of both point-to-point and collective communications, which is not the case with regular profiling libraries. Indeed, thanks to this low-level monitoring, we can see the control messages as well as the implementation-specific messages forwarded during a collective operation (e.g., during a gather or a scatter). Since this monitoring is very light, it does not disturb the application execution.

The main drawback of this approach is the following: this preliminary run of the application is mandatory and a change in the execution (e.g., the number of processors, the input data, etc.) often leads to a rerun of the profiling. However, this step is necessary for legacy MPI applications for which the pattern is not already available. Indeed, newly developed MPI applications could provide the communication pattern directly to the relevant MPI routine (see Step 3 in Section 2.3). In such a case, this first step is not required anymore. We believe that this monitoring approach is relevant in many cases. Indeed, there are classes of scientific applications that possess a *static* communication pattern. For instance, CFD applications feature a regular pattern that is repeated at each step of the algorithm.

For now, we consider only a *spatial* pattern, that is, we do not take into consideration the changes (if any) in the application behavior during its execution. Indeed, we consider that the pattern is *static* and we aim to optimize the placement based on the behavior of the whole execution of the application. Also, we consider *static* applications, where the number of MPI processes is constant during their execution. We derive several metrics from the generated trace:

msg is the number of messages exchanged between pairs of MPI processes. Such a view is important when we have a lot of small messages and communications are latency-bound.

1. These monitoring elements account for about a hundred lines of code in the MPICH2 or Open MPI software stacks for instance.

size is the amount of data exchanged between pairs of MPI processes. Such a view is important for bandwidth-bound communications in the application.

avg is the average size of the messages exchanged between pairs of MPI processes.

2.2 Next Step: Modeling the Hardware Architecture

The second step to determine a relevant process placement is to retrieve information about the underlying hardware (e.g., memory hierarchy, cores numbers, etc.). To achieve this in a portable way is not straightforward. Actually, until recently, no tool was able to easily provide information about the various cache levels (such as their sizes and which cores access them) on a wide range of systems. To this end, we participated in the development of a software to fulfill this goal: Hardware Locality or HWLOC [10].

Indeed, thanks to HWLOC the hardware architecture can be modeled by a tree, the depth of which corresponds to the depth of the hardware component in the hierarchy (e.g., network switches, cabinet, nodes, processors, caches, cores) and where the leaves are the computing units of the architecture. HWLOC allows us to model the architecture in a portable fashion (i.e., across operating systems). It is also flexible: this modeling can be performed dynamically because HWLOC is implemented as a library that is callable by another software, such as an MPI implementation.

However, the use of HWLOC in our work is not mandatory. Actually, one of our previous work [11] relied on a topology matrix[2] to model the architecture. Because such a representation induces a flattening of the view of the hardware structure, valuable information that could be exploited by the matching algorithm is lost. Moreover, since a NUMA node is most of the time hierarchically structured, a tree provides a more reliable representation than a topology matrix.

2.3 Last Step: Computing and Enforcing the Process Placement

In this section, we describe how to enforce the *placement policy* determined by the matching algorithm after both previous pieces of information have been gathered. Our algorithm is exposed in Section 4 (and further detailed in Appendix D.1, available in the online supplemental material). Enforcing the placement policy means that each MPI process has to be executed on its own dedicated computing unit. This task is out of the scope of the MPI standard and falls on the MPI implementation process manager or runtime system.

There are two methods to enforce the process placement policy. The first one is the *resource binding* technique [11], [13]. Generally speaking, binding the processes of a parallel application to computing units leads to a decrease of the system noise and improves performance (see Appendix B, available in the online supplemental material). In this case, the matching algorithm computes on which physical computing unit an MPI process should be located. Therefore, a unique MPI process rank of the application corresponds to a single computing unit number.[3] The MPI implementation process manager then binds the application processes accordingly. Legacy MPI applications do not need to be modified to take advantage of this approach. Its drawbacks are its lack of transparency because the user has to rely on MPI implementation-specific options and its lack of flexibility since changing the binding during an application execution is difficult.

The second method is called *rank reordering*. In this case, the MPI processes are first bound to computing units when the application is launched, but without following a specific or optimized binding policy. Then, the MPI application creates a new communicator with application-specific information attached to it. The ranks of the MPI processes belonging to this communicator can be *reordered*, that is, changed to fit some application constraints. In particular, these rank numbers can be modified to create a match between the application communication pattern and the underlying physical architecture. In this case, the matching algorithm computes a new MPI rank number for each process rather than a resource number. This reordering of rank numbers should be performed before any application data are loaded into the MPI processes in order to avoid data movements afterwards.

Legacy MPI applications need to be modified to issue a call to a rank-reordering MPI function and then use the new communicator. Fortunately, the extent of these modifications is quite limited (a few dozen code lines, see Appendix C, available in the online supplemental material, for examples). `MPI_Dist_graph_create` (part of the standard since MPI 2.2 [4]) is one such MPI function with rank reordering capabilities. It takes as arguments a set of pointers (`sources`, `destinations`, `degrees` and `weights`) that define a graph. These pointers can convey random application communication patterns to the MPI implementation. As the current implementation of this function in Open MPI and MPICH2 software stacks does not perform any rank reordering, we improved both versions by integrating HWLOC and our TREEMATCH matching algorithm. Mercier and Jeannot [6] describes a *centralized* version of this work in which a single MPI process gathers all the hardware information with HWLOC, then calls TREEMATCH to compute the reordering and finally broadcasts the new ranks to all the other MPI processes of the application. To circumvent the lack of scalability of this approach, we implemented, for this paper, a *partially distributed* version in which each node reorders only its (local) MPI processes. In this case, scalability is improved but the initial dispatch of MPI processes has influence on the final result. Indeed, the processes running TREEMATCH possess local information only and therefore cannot reduce the amount of internode communication in the application. Relying on a standard MPI call ensures portability, transparency and dynamicity as it can be issued multiple times during an application execution. These aspects aside, the rank reordering technique yields the same performance improvements as the resource binding technique.

2. The same approach as in [12].

3. We make the hypothesis that there is no oversubscribing of the computing units.

3 State of the Art

The issue of process placement on processors in order to match a communication pattern to the underlying hardware architecture has been studied previously. This mapping problem is usually modeled as a *graph embedding problem*. More precisely, the problem is introduced in [14] and an algorithm based on the Kernighan-Lin heuristic [15] is described as well as results for several benchmarks. However, this work is tailored for a specific vendor hardware and is thus not suitable for generic architectures. Also, the author optimizes some of the routines that create *Cartesian topologies* but leaves unaddressed the generic *graph topology* case. The experiments show dramatic improvements but are restricted to benchmarks that only perform communications and no computation.

Some other works address generic virtual topologies (used to express the communication pattern) but consider only the network physical topology for the hardware aspects. The Blue Gene class of machines has been especially targeted [16], [17], or [18]. InfiniBand fabric is also a subject of studies: [19] and [20] empirically assess the performance of the interconnection network to provide a usable model of the underlying architecture. Subramoni et al. [21] use the Neighbor Joining Method to detect the physical topology of the underlying InfiniBand network. LibTopoMap [22] also considers generic network topologies and relies on ParMETIS [23] to solve the resulting graph problem. Such approaches are definitively complementary to our work as they do not take into account the internal structure of multicore nodes.

MPI topology mechanism implementation issues are discussed in [24]. Both Cartesian and graph topologies are addressed by this work, and the algorithm proposed is also based on the Kernighan-Lin heuristic. The optimization criterion considered is either the total communication cost or the optimal load balance. Again, this work is designed for a specific vendor hardware (NEC SX series). The proposed approach is thus less generic than ours and, more importantly, does not apply to clusters of multicore nodes, which are our target architectures.

MPIPP [12] is a set of tools aimed at optimizing an MPI application execution on the underlying hardware. MPIPP relies on an external tool to gather the hardware information statically, while we manage to perform this task dynamically at runtime (see Section 2.2). Also, MPIPP allows only the dispatching of MPI processes on nodes (machines) and does not address the mapping of processes on specific computing units within a node. Multicore machines are thus not fully exploited, as the memory hierarchy cannot be taken into account. The same drawback applies to [25], which manages to effectively reduce the amount of internode communication of an MPI application by performing a so-called reordering operation. However, the meaning of *reordering* in [25] is different from our work. Indeed, [25] only reorganizes the file containing the node names (a.k.a the hosts file), thus changing the way processes are dispatched on the nodes. That is, the MPI processes are not bound to dedicated computing units and the application does not actually call any real MPI reordering routine. Hence, our partially distributed implementation of `MPI_Dist_graph_create` could be an ideal complement to this work.

Some recent works bind MPI processes to dedicated computing units in order to improve communications performance. This *resource binding* technique is studied in [13] and [11]. Both were published around the same time and share a very similar design. As a consequence, they suffer from the same limitations: they do not make use of standard MPI calls to reorder the process ranks, they both rely on the Scotch [26] partitioner and they are unable to gather the hardware information dynamically at runtime. Also, [13] uses a purely quantitative approach, while ours is qualitative since we manage to use the *structure* of the memory hierarchy of a node. It is worth noting that major free MPI implementations such as MPICH2 [27] and Open MPI [28] provide options to perform this binding of processes at launch time thanks to their process managers, Hydra and ORTE, respectively. The user can choose from some more or less sophisticated predefined placement policies (e.g., [29]). However, such policies are generic and fail to consider the application's communication patterns specificities. There are other runtime systems that make use of the resource binding technique, such as the ones provided by MPI vendors' implementations from Cray [30], [31], HP [32] and IBM (according to [33]).

Collective communications are an important feature of the MPI interface and several works aim at to improve their performance by taking into account the underlying physical architecture. For instance, [34] uses a hierarchical two-level scheme to make better use of multicore nodes. Zhang et al. [35] and Ma et al. [36] introduce process placement strategies in collectives to find the most suitable algorithm for the considered collective operation. Our work considers all communication in the application and is not restricted to collective operations.

Besides the aforementioned Kernighan-Lin heuristic, there are algorithms that are able to solve a *Graph Embedding Problem*. Chaco [37] and Metis [23] (or ParMetis for its parallel version) are examples of such graph-partitioning software. Scotch [26] is a graph-partitioning framework that is able to deal with tree-structured input data (called *tleaf*) to perform the mapping. An important difference between graph partitioning/embedding techniques and our work is that we only need the structure and the topology of the target hardware while the other works require quantitative information about the hardware in order to make a precise evaluation of the communication time, which is, most of the time, impossible to collect on current hardware due to NUMA effects.

Programming models other than MPI can be used to address the problem of process placement. For instance, in CHARM++ [7], it is possible to perform dynamic load balancing of internal objects (chares) using information about the affinity and the topology. PGAS languages (e.g., UPC [38]) expose a simple two-level scheme (local and remote) for memory affinity that can be used for mapping processes.

4 The TreeMatch Algorithm

In this section, we present our matching algorithm, called TreeMatch. This algorithm, depicted in Algorithm 1, is able

to compute a matching for the *resource binding* technique (i.e., computing units numbers) or for the *rank reordering* technique (i.e., new MPI ranks). A first version of TreeMatch was published in [5]. More details about TreeMatch are given in Appendix D.1, available in the online supplemental material.

Algorithm 1: The TreeMatch Algorithm

```
Input: T// The topology tree
Input: m // The communication matrix
Input: D // The depth of the tree
groups[1..D − 1]=∅ // How nodes are grouped on each level
foreach depth← D − 1..1 do // We start from the leaves
    p ← order of m
    // Extend the communication matrix if necessary
    if p mod arity(T, depth − 1) ≠ 0 then
        m ←ExtendComMatrix(T,m,depth)
    groups[depth]←GroupProcesses(T,m,depth)// Group
    processes by communication affinity
    m ←AggregateComMatrix(m,groups[depth])     // Aggregate
    communication of the group of processes
MapGroups(T,groups) // Process the groups to built the
mapping
```

TreeMatch uses a tree for modeling the hardware, while other solutions (e.g., MPIPP, ParMetis, etc.) need a topology matrix describing the communication cost between each pair of processes. Having a tree eases significantly the algorithmic process by ignoring the quantitative aspect of the communication: the speed and latency of the cache hierarchy. Indeed, gathering such information is not always easy and communication speeds between computing units are very hard to model accurately as they depend on many factors (message size, cache size, contention, latency, bandwidth, etc.). Therefore, using only structural and qualitative information avoids such inaccuracy and enables more portable solutions based only on the target hardware structure (see Appendix E, available in the online supplemental material, for more details).

In the case where the nodes allocated to the applications are scattered all over the parallel machine, TreeMatch is still able to provide a solution. If the network topology of the machine is a tree, it must abstract the allocated portion using a balanced tree covering it (perhaps at the cost of flattening the structure). If the topology is an arbitrary graph, the network needs to be abstracted by a single node in the topology. In Appendix G.1, available in the online supplemental material, we provide an experiment that shows that in some cases, flattening the network does not significantly hinder the performance. Moreover, as explained in the state-of-the-art Section, complex network topologies are addressed by other works in a complementary fashion.

In TreeMatch, the topology tree is processed upward and processes are recursively grouped according to the arity of the next considered level. The main cost of the algorithm is in the function `GroupProcesses` (see Appendix D.1, available in the online supplemental material). If k is the arity of the next level and p is the order of the current communication matrix, the complexity of this part is proportional to the number of k-sized groups among a set of p elements, and this number is $\binom{p}{k}$. However, $\binom{p}{k} = O(p^k)$. Hence, the standard version has an exponential complexity.

To avoid combinatorial explosion of the algorithm running time we provide, in this paper, two new mechanisms. First, we artificially decompose a level of the tree into several levels. Second, we simplify the search and building of groups when the number of such groups is large. Both optimizations are detailed and discussed in the following subsection.

4.1 Running Time Optimization of TreeMatch

4.1.1 Arity Division of the Tree

In order to reduce the number of possible groups in the function `GroupProcesses`, we decompose a level of the tree of high arity into one or several levels of smaller arity. The only constraint is that the product of the arities of these new levels has to be equal to the arity of the original level.

For instance, if a tree has a level with an arity of 4, we decompose this level into two levels of arity 2. This increases the number of times the function `GroupProcesses` is called. But, as $2 \times \binom{p}{2}$ is smaller than $\binom{p}{4}$ for $p > 8$, this is beneficial as long as we have to map eight processes or more (i.e., we are dealing with lower levels of the tree).

More generally, we can decompose k into prime factors and compute a decomposition of a node of arity k into different levels with an arity corresponding to each factor. As in modern computers the number of computing units in a node is generally a multiple of 2 or 3, such techniques help to reduce k to reasonable values for the lower levels of the tree (when p is high).

Let $f_{k,d}(p) = \frac{\binom{p}{k}}{d\binom{p}{k/d}}$, the function that models the gain when we divide a node of arity k into d nodes of arity k/d for p processes ($k < p$ and d divides k). Let us study when we have a gain (i.e., when $f_{k,d}(p) > 1$).

By definition of $\binom{p}{k}$, we have:[4]

$$f_{k,d}(p) = \frac{\left(\frac{k}{d}\right)!\left(p-\frac{k}{d}\right)!}{k!(p-k)!d}.$$

The first order derivative is:

$$f'_{k,d}(p) = -\frac{\left(\Psi(p-k+1) - \Psi\left(\frac{pd-k+d}{d}\right)\right)\left(\frac{pd-k}{d}\right)!\left(\frac{k}{d}\right)!}{k!(p-k)!d},$$

where Ψ is the *Digamma* function that increases in $]0, +\infty[$. As $p-k+1 < \frac{pd-k+d}{d}$, $f'_{k,d}(p)$ is positive for $0 < k < p$. Hence, $f_{k,d}(p)$ is increasing with p. Moreover,

$$\lim_{p\to\infty} f_{k,d}(p) = \frac{\left(\frac{k}{d}\right)!}{k!d} \lim_{p\to\infty} \frac{\left(p-\frac{k}{d}\right)!}{(p-k)!} = +\infty,$$

because $k < p$ and $d > 1$. Therefore, there exists a number p^*, such that $\forall p > p^*, f_{k,d}(p) > 1$ and hence $\binom{p}{k} > d\binom{p}{k/d}$. This means that for any given node of arity k, there is a number of processes above which it is always better to decompose this node in d nodes of arity k/d (where d divides k).

Based on this consideration, we now ask this question: given a node of arity k, what is the value of p^* and what is the optimal value (d^*) of d? In order to answer this question, we have computed all the possibilities for $k \leq 128$ and

4. Most of these computations can be checked using Matlab.

$p \leq 500{,}000$. The best value of d is the one that minimizes $d\binom{p}{k/d}$. Interestingly enough, it appears that for all tested values of k and p this optimal value d^* does not depend on p and is always the greatest non trivial divisor of d (i.e., the greatest divisor not equal to d). This means that for any value of k there is only one value of p^* and one value of d^* such that $\forall p \geq p^*, \binom{p}{k} > d^*\binom{p}{k/d^*}$ and $\forall p \geq p^*, d \leq d^*, d\binom{p}{k/d} > d^*\binom{p}{k/d^*}$. For all $k \leq 128$ and not prime, we display the values of p^* and d^* in Table 2 found in Appendix D.2, available in the online supplemental material.

Given a tree, it is now easy to optimally decompose it into a tree for which nodes of high arity are decomposed into nodes of smaller arity. We first recall that in this work we assume that the arity of all the nodes of a given level of a tree is the same. Then, at level n given a node of arity k_n, in order to decide if we decompose it or not we need to compute p, the number of processes (or group of processes) that will be considered by the TreeMatch algorithm. We have $p = \prod_{i=0}^{n} k_i$: the number of nodes of the considered trees of a given level is the product of the arities of the above levels. Then, if in the table at row k, p is greater than p^*, we divide all the nodes of the level into d^* nodes of arity k/d^*. To deal with large arities, we traverse the tree several times (to check if the new node of arity K/d^* can also be decomposed) until there is no more possible node decomposition.

For example, consider a tree of depth 3 with arity from root to leaves equal to 4, 4 and 1. Based on Table 2 of Appendix D.2, available in the online supplemental material, we see that it is optimal to only decompose the four nodes of the second level because, for the first level, the TreeMatch algorithm will deal only with four groups of processes. After optimization of the tree, we obtain four levels with arities of 4, 2, 2 and 1.

4.1.2 Speeding up the Group Building

Reducing the arity of a node is very useful as $\binom{p}{k} = O(p^k)$. Thanks to the above techniques, most of the current architectures can be decomposed in trees with arities 2 and/or 3, reducing the complexity of each TreeMatch step to squared or cubic complexities. However, even in this case, the cost of these steps can be very high if we want to use TreeMatch at runtime (e.g., in a load-balancer). Moreover, we cannot take this actual state for granted, and it is possible that internal arities of nodes will be higher in the future. For instance, it is already the case in some machines that the current arity of network switches is neither a multiple of 2 nor 3.

In order to handle this case, we have introduced a faster way of grouping processes or groups of processes. This is described in the `FastGroupProcesses` function, which is executed instead of the `GroupProcesses` one when $\binom{p}{k} \geq T_h$, where T_h is a user-given threshold (30,000 by default).

Function FastGroupProcesses(T,m,depth)

Input: b //Number of buckets
Input: T //The topology tree
Input: m // The communication matrix
1 bucket_list←PartialSort(m, b);
2 **return** GreedyGrouping(bucket_list,T)

It works as follows: first, elements of the matrix are sorted according to their orders of magnitude into b buckets (there are eight buckets by default). The largest elements of the matrix are put in the first bucket, smaller elements in the last bucket. To construct these buckets, we randomly extract a sample of 2^b elements of the matrix. Then, we sort this sample. To perform the partial sorting we extract $b-1$ pivots from this sample. The first pivot is the largest element of the sample and the ith pivot p_i is the 2^{i-1} largest element of the sample. Then, we set $p_0 = +\infty$ and $p_b = 0$. Then, each element of the matrix of value v is put in the bucket j such that $p_{j-1} < v \leq p_j$.

Once all the matrix elements are put in the list of buckets, we consider these elements bucket by bucket, starting with the bucket of largest elements. We sort the current bucket and we group the largest elements of the current bucket together while there are not enough groups. This is done greedily with the only constraint being that an element of the matrix cannot be in two different groups. If a bucket is exhausted, we take the next one.

5 Experimental Validation

In this section, we detail both the hardware and software elements used in our experiments and we analyze the results achieved.

5.1 Experimental Environment

All experiments have been carried out on a cluster called PlaFRIM. This cluster is composed of 64 nodes linked with an InfiniBand interconnect (HCA: Mellanox Technologies, MT26428 ConnectX IB QDR). Each node features two Quad-core- Intel Xeon Nehalem X5550 (2.66 GHz) processors. Eight Mbytes of L3 cache are shared between the four cores of a CPU. There are also 24 GB of 1.33 GHz DDR3 RAM on each node. The operating system is a SUSE Linux (2.6.27 kernel). We reserved two InfiniBand QDR switches and 16 nodes on each to perform the experiments. As for the software, by default we used Open MPI ver. 1.5.4 (MVAPICH2 ver 1.8 for one experiment) and Hwloc ver. 1.4.1.

First, we ran experiments with the NAS Parallel Benchmarks [39]. We focused on three particular *kernels*:

- the conjugate gradient (CG) kernel because of its irregular memory accesses and communications
- the fourier transform (FT) kernel for its all-to-all communication pattern
- the Lower-Upper Gauss-Seidel kernel (LU), which features a solver with irregular memory accesses

For these three kernels we used two *classes* (C and D) to represent average or large problem sizes. We also chose to test process placement on a real-world application: ZEUS-MP/2 [40]. ZEUS-MP/2 is a CFD application that includes gas hydrodynamics, ideal magnetohydrodynamics, flux-limited radiation diffusion, self gravity, and multispecies advection.

We compared TreeMatch with Scotch, ParMETIS, Chaco and MPIPP. As MPIPP is a randomized strategy we have two versions: MPIPP1 and MPIPP5. MPIPP5 consists of applying MPIPP1 five times. We also tested two greedy process placement policies. The first one, called Round Robin

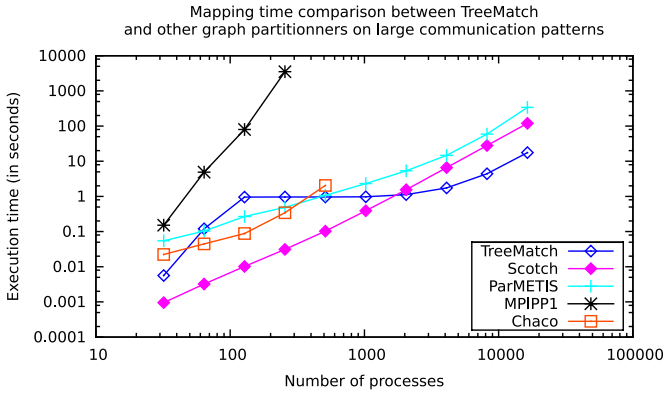

Fig. 1. Average mapping computation time comparison for various placement methods.

(*RR*), corresponds to the *physical identity* (process i is mapped onto *physical* computing unit i).[5] The second one, called *Packed*, corresponds to the *logical identity* (process i is mapped onto *logical* computing unit i). Logical numbering is usually different from the physical one: in logical numbering, units are numbered consecutively using a breadth-first search traversal of the tree.[6] Some partitioners are natively able to find a solution to the process mapping problem as defined here (e.g., Scotch with the tleaf input data, MPIPP). For ParMETIS and Chaco, we implemented a graph-embedding algorithm to solve the mapping problem by leveraging their k-way partitioning capabilities. It is worth noting that because of a coarsening algorithm, Scotch and ParMETIS sometimes need a normalized matrix.

Another important issue encountered with Scotch is the following: it takes as input a *tleaf* file to represent the architecture with an edge-weighted tree. These weights are used to compute the process placement and the resulting mapping depends on these values. In our results, we used two versions of Scotch: the first one, called *Scotch*, uses very small values (between 1 and 4) for the weights while the second one, called *Scotch_w*, uses larger values (between 10 and 500). See Appendix F, available in the online supplemental material, for a detailed discussion.

We used the three metrics described in Section 2.1: the number of messages exchanged (msg), the amount of data exchanged (size) and a value corresponding to the average size of one message (avg). As for the processes count, we ran every test case with 64, 128 and 256 process configuration (and the same number of computing units).

5.2 Results

5.2.1 Mapping Computation Time

In this section, we measured the mapping computation time of each graph partitioner and TreeMatch. We mapped a communication graph ranging from 64 to 16,384 vertices (corresponding to the same number of processes) on a topology tree modeling 128 switches of 16 nodes with two quad-core sockets. These communication graphs are dense graphs, modeling patterns where all processes communicate at least once with every other one.

Results are depicted in Fig. 1. It shows the average runtime of 10 executions versus the graph size. Only calculation times are displayed. I/O timings (i.e., loading the graph and writing the solution to disk) are excluded. Runs have been made on a 2.66 GHz Intel Nehalem CPU.

We excluded the mapping time of MPIPP5. As we shall see in the results, MPIPP1 is already the slowest placement method (more than 3,550 s on a 256-vertices graph) and MPIPP5 is on average five times slower than MPIPP1. We do not plot the Chaco graph after a size of 512 because it fails to handle larger graphs.

On this plot, we can see that for small cases, Scotch is the fastest solution. For a 128-vertices graph, Scotch takes 10 ms while TreeMatch takes 957 ms. However, beyond this size, TreeMatch changes its mapping strategy (as explained in Section 4.1.2) and the slope of its curve flattens. For size 2,048 and more, TreeMatch is the fastest strategy. For size 16,384, TreeMatch is seven times faster than Scotch and 20 times faster than ParMETIS. This demonstrates that TreeMatch scales better than the other methods.

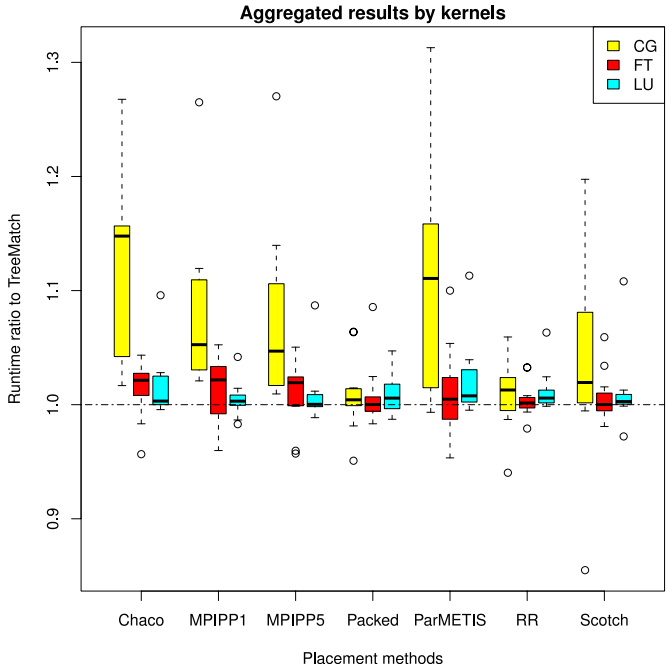

Fig. 2. Average execution time ratio between TreeMatch and other placement methods for the NAS benchmarks. Results projected by kernels (LU, CG and FT). Metric Avg excluded.

5.2.2 NAS Parallel Benchmarks Comparison

For our experiments, we did a Cartesian product of all variable parameters (i.e., metrics, kernels, classes and process counts) leading to $3 \times 3 \times 2 \times 4 = 72$ cases. Each case was run ten times and we computed the average execution time.

Fig. 2 shows the projection for the various NAS kernels (i.e., CG, FT and LU) and depicts the ratio to TreeMatch for each placement method using boxplots: the higher the ratio, the better TreeMatch is. Each boxplot graphically presents five statistics:[7] the median (bold line), the lower

5. This policy is typically enforced by batch schedulers when reserving nodes in a cluster for MPI applications.

6. In Appendix D.1, available in the online supplemental material, Fig. 9 depicts the difference between *RR* and *Packed* policies.

7. See http://en.wikipedia.org/wiki/Box_plot for more details.

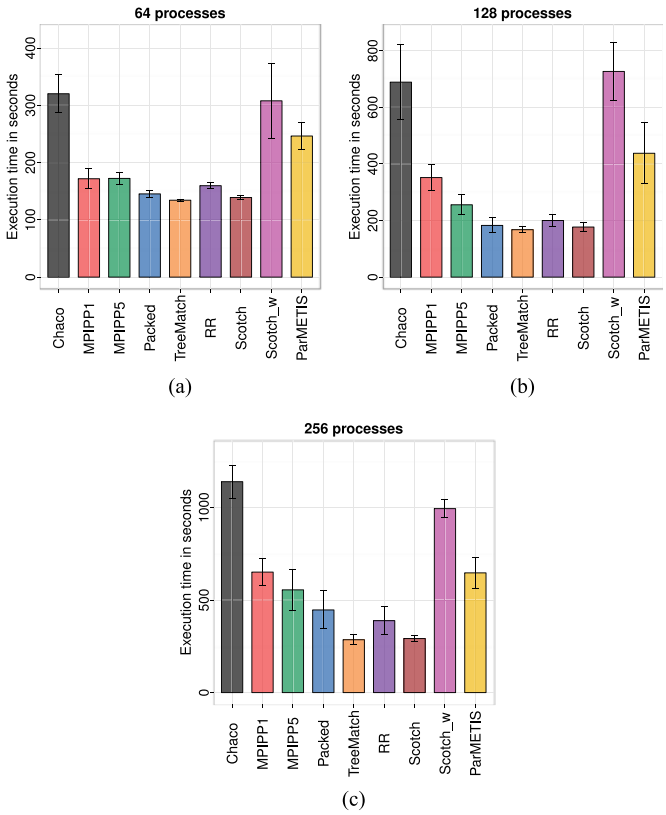

Fig. 3. Average execution time of ZEUS-MP/2 on several numbers of processes (average of 10 runs, *msg* metric, 3,000 iterations).

and upper quartile (colored box), lower (resp. upper) whiskers represent the lowest (resp. largest) datum within 1.5 times the interquartile range of the lower (resp. upper) quartile, outliers are shown as dots.

In this experiment, as there is no difference between the two versions of Scotch (small or large weights), we only show the small weights version.

On average, we see that *Packed* and *RR* are outperformed by TreeMatch. This is due to the fact that *RR* and *Packed* are efficient only for communication matrices that have large elements near the diagonal.

TreeMatch shows better performance than the other methods. The best gain is achieved for the CG kernel. This is explained by the fact that the communication matrix is highly irregular and with large communication outside of the diagonal. Therefore, the placement proposed by TreeMatch greatly reduces the costs associated with these communications. For the FT kernel the gains are small because the communication matrix is very homogeneous (especially for the size metric). Hence, the process placement has only a moderate influence on the execution time. The LU kernel is between the CG and the FT kernels in terms of regularity and diagonal dominance and small gains are achievable with this kernel. More results can be found in Appendix G.2, available in the online supplemental material.

5.2.3 ZEUS-MP/2 Comparison

In this section we present experiments carried out on the PlaFRIM platform with the ZEUS-MP/2 CFD application. We

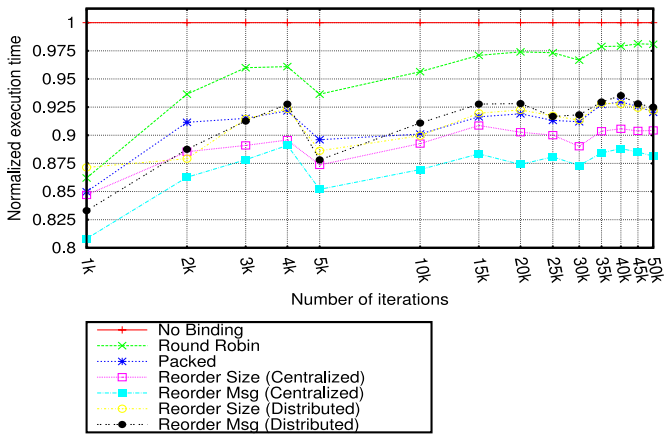

Fig. 4. ZEUS-MP/2 (mhd blast case, 64 processes, MVAPICH2): influence of the placement policy on performance.

have chosen to show the msg metric as it leads to the lowest execution time for any method and for up to 3000 iterations.

Figs. 3a, 3b, and 3c depict the results for different process counts.

The ZEUS-MP/2 communication pattern is very irregular and process placement impacts the execution time. *RR* and *Packed* also yield good results and rank third and fourth in this experiment. Moreover, for 256 processes TreeMatch outperforms *RR* by more than 25 percent (285.62 s versus 388.61 s). Other methods lead to longer execution times, especially graph partitioners such as Chaco or ParMETIS.

For these experiments, we see a difference whenever Scotch uses small or large weights for describing the topology. The performance of the large weight case is the worst. For us, finding the best weights has not been possible without testing the mapping produced by Scotch and the slight difference between the two configurations leads to a very large difference in terms of performance (timings are more than doubled in Fig. 3c). TreeMatch does not suffer from this drawback as it relies only on structural properties of the topology. Moreover, as shown in Fig. 11 of Appendix E, available in the online supplemental material, the communication time between computing units is not a linear (not even an affine) function of the message size. This explains why the tleaf model used by Scotch is not able to capture the time taken to send a message.

5.2.4 Centralized versus Distributed Mapping

Fig. 4 shows the performance improvements obtained for ZEUS-MP/2 (mhd blast case, 64 processes) for various placement policies. Our reference policy is *RR*. Besides *Packed*, we tested TreeMatch with size and msg metrics, both in *centralized* and *partially distributed* ways (c.f. Section 2.3). The results confirm that the best metric is msg and show that centralized reordering performs better than other policies. This is due to the fact that we manage to reduce internode traffic and to improve intranode communication. For larger iteration counts, the gain is roughly 12 percent (67 s execution time for *RR* versus 59 s). As for the partially distributed reordering, the initial dispatch of processes was similar to that of *RR*. Hence, even if the performance delivered is only on a par with *Packed*, we still

manage to improve on *RR*. But as the machines feature only eight cores, we cannot expect more improvements. We would like to make tests on nodes featuring larger numbers of cores in the future.

6 Conclusion and Future Works

The locality problem is becoming a major challenge for current applications. Improving data access and communication is a key issue for obtaining the full performance of the underlying hardware. However, not only does the communication speed between computing units depend on their locations (due to cache size, memory hierarchy, latency and network bandwidth, topology, etc.), but also the communication pattern between processes is not uniform (some pairs of processes exchange far more data than others).

In this paper, we have presented a new algorithm called TreeMatch, which computes a process placement of the application tailored for the target machine. It is based on both the application communication pattern and the architecture of the machine. Unlike other approaches using graph-partitioning techniques, TreeMatch does not need accurate and quantitative information about the various communication speeds. Moreover, to speed up the algorithm, we have proposed several optimizations: one is based on tree decomposition while the other relies on a fast group building.

To evaluate our solution, we have compared TreeMatch against state-of-the-art strategies. Two kinds of applications have been tested: the NAS parallel benchmarks and a CFD application (ZEUS-MP/2). Results show that TreeMatch consistently outperforms graph-partitioning based techniques. Regarding the *Packed* and *RR* policies, the more irregular the communication pattern, the better the results. Gains of up to 25 percent have been exhibited in some cases.

TreeMatch is available in several implementations of MPI (MPICH2 and Open MPI) as the `MPI_Dist_graph_create` function enabling rank reordering. Moreover, a partially distributed version for large NUMA nodes interconnected by a network is also provided.

Future works are directed towards a fully distributed version of TreeMatch for the use of large-scale machines. Another study will focus on easing the gathering of the communication pattern. Several ways are possible, from static analysis of the code to simulation of the communications or user-given information based on the structure of the data. Experiments on very large machines are also targeted, especially ones with a large number of cores. This might require even further optimization of TreeMatch.

References

[1] J. Dongarra and P. Beckman et al., "The International Exascale Software Roadmap," *Int'l J. High Performance Computer Applications*, vol. 25, no. 1, pp. 3-60, 2011.

[2] PRACE, "The Scientific Case for High Performance Computing in Europe," report, http://www.prace-ri.eu/IMG/pdf/prace_the_scientific_case_executive_s.pdf, p. 147, 2012.

[3] *MPI: A Message-Passing Interface Standard*, Message Passing Interface Forum, Mar. 1994.

[4] T. Hoefler, R. Rabenseifner, H. Ritzdorf, B.R. de Supinski, R. Thakur, and J.L. Träff, "The Scalable Process Topology Interface of MPI 2.2," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 4, pp. 293-310, 2011.

[5] E. Jeannot and G. Mercier, "Near-Optimal Placement of MPI Processes on Hierarchical NUMA Architectures," *Proc. 16th Int'l Euro-Par Conf. Parallel Processing (Euro-Par '10)*, pp. 199-210, Sept. 2010.

[6] G. Mercier and E. Jeannot, "Improving MPI Applications Performance on Multicore Clusters with Rank Reordering," *Proc. 18th European MPI Users' Group Conf. Recent Advances in the Message Passing Interface (EuroMPI)*, pp. 39-49, Sept. 2011.

[7] L. Kale and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," *Proc. Eighth Ann. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA' 93)*, pp. 91-108, Sept. 1993.

[8] C. Ma, Y.M. Teo, V. March, N. Xiong, I.R. Pop, Y.X. He, and S. See, "An Approach for Matching Communication Patterns in Parallels Applications," *Proc. 23rd IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS '09)*, May 2009.

[9] D. Buntinas, G. Mercier, and W. Gropp, "Implementation and Evaluation of Shared-Memory Communication and Synchronization Operations in MPICH2 Using the Nemesis Communication Subsystem," *J. Parallel Computing, Selected Papers from EuroPVM/MPI 2006*, vol. 33, no. 9, pp. 634-644, Sept. 2007.

[10] 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," *Proc. 18th Euromicro Int'l Conf. Parallel, Distributed and Network-Based Processing (PDP '10)*, http://hal.inria.fr/inria-00429889, Feb. 2010.

[11] G. Mercier and J. Clet-Ortega, "Towards an Efficient Process Placement Policy for MPI Applications in Multicore Environments," *Proc. 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface (EuroPVM/MPI)*, pp. 104-115, Sept. 2009.

[12] H. Chen, W. Chen, J. Huang, B. Robert, and H. Kuhn, "MPIPP: An Automatic Profile-Guided Parallel Process Placement Toolset for SMP Clusters and Multiclusters," *Proc. 20th Ann. Int'l Conf. Supercomputing (ICS)*, pp. 353-360, 2006.

[13] E. Rodrigues, F. Madruga, P. Navaux, and J. Panetta, "Multicore Aware Process Mapping and Its Impact on Communication Overhead of Parallel Applications," *Proc. IEEE Symp. Computers and Comm.*, pp. 811-817, July 2009.

[14] T. Hatazaki, "Rank Reordering Strategy for MPI Topology Creation Functions," *Proc. Fifth European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, vol. 1497, pp. 188-195, http://dx.doi.org/10.1007/BFb0056575, 1998.

[15] B.W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell System Technical J.*, vol. 49, no. 2, pp. 291-307, Feb. 1970.

[16] B.E. Smith and B. Bode, "Performance Effects of Node Mappings on the IBM BlueGene/L Machine," *Proc. European Conf. Parallel Processing (Euro-Par)*, pp. 1005-1013, 2005.

[17] H. Yu, I.-H. Chung, and J.E. Moreira, "Blue Gene System Software-Topology Mapping for Blue Gene/L Supercomputer," *Proc. Int'l Conf. Supercomputing (SC)*, p. 116, 2006.

[18] P. Balaji, R. Gupta, A. Vishnu, and P.H. Beckman, "Mapping Communication Layouts to Network Hardware Characteristics on Massive-Scale Blue Gene Systems," *Computer Science - R&D*, vol. 26, no. 3/4, pp. 247-256, 2011.

[19] M.J. Rashti, J. Green, P. Balaji, A. Afsahi, and W. Gropp, "Multi-Core and Network Aware MPI Topology Functions," *Proc. 18th European MPI Users' Group Conf. Recent Advances in the Message Passing Interface (EuroMPI)*, pp. 50-60, 2011.

[20] S. Ito, K. Goto, and K. Ono, "Automatically Optimized Core Mapping to Subdomains of Domain Decomposition Method on Multicore Parallel Environments," *Computer & Fluids*, vol. 80, pp. 88-93, Apr. 2012.

[21] H. Subramoni, S. Potluri, K. Kandalla, B. Barth, J. Vienne, J. Keasler, K. Tomko, K. Schulz, A. Moody, and D. Panda, "Design of a Scalable Infiniband Topology Service to Enable Network-Topology-Aware Placement of Processes," *Proc. ACM/IEEE Conf. Supercomputing (CDROM)*, p. 12, 2012.

[22] T. Hoefler and M. Snir, "Generic Topology Mapping Strategies for Large-Scale Parallel Architectures," *Proc. Int'l Conf. Supercomputing (ICS)*, pp. 75-84, 2011.

[23] G. Karypis and V. Kumar, "METIS - Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0," technical report 1995.

[24] J.L. Träff, "Implementing the MPI Process Topology Mechanism," *Proc. ACM/IEEE Conf. Supercomputing (Supercomputing '02)*, pp. 1-14, 2002.
[25] B. Brandfass, T. Alrutz, and T. Gerhold, "Rank Reordering for MPI Communication Optimization," *Computer & Fluids*, vol. 80, pp. 372-380, Jan. 2012.
[26] F. Pellegrini, "Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs," *Proc. Scalable High-Performance Computing Conf. (SHPCC '94)*, pp. 486-493, May 1994.
[27] Argonne National Laboratory, "MPICH2," http://www.mcs.anl.gov/mpi/2004, 2013.
[28] E. Gabriel, G.E. Fagg, G. Bosilca, T. Angskun, J. Dongarra, J.M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R.H. Castain, D.J. Daniel, R.L. Graham, and T.S. Woodall, "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation," *Proc. 11th European PVM/MPI Users' Group Meeting*, pp. 97-104, Sept. 2004.
[29] J. Hursey, J.M. Squyres, and T. Dontje, "Locality-Aware Parallel Process Mapping for Multi-Core HPC Systems," *Proc. IEEE Int'l Conf. Cluster Computing (CLUSTER)*, pp. 527-531, 2011.
[30] Nat'l Inst. for Computational Sciences, "MPI Tips on Cray XT5," http://www.nics.tennessee.edu/user-support/mpi-tips-for-cray-xt5, 2013.
[31] J.L. Whitt, G. Brook, and M. Fahey, "Cray MPT: MPI on the Cray XT," http://www.nccs.gov/wp-content/uploads/2011/03/MPT-OLCF11.pdf, 2011.
[32] D. Solt, "A Profile Based Approach for Topology Aware MPI Rank Placement," http://www.tlc2.uh.edu/hpcc07/Schedule/speakers/hpcc_hp-mpi_solt.ppt, 2007.
[33] E. Duesterwald, R.W. Wisniewski, P.F. Sweeney, G. Cascaval, and S.E. Smith, "Method and System for Optimizing Communication in MPI Programs for an Execution Environment," http://www.faqs.org/patents/app/20080288957, 2008.
[34] H. Zhu, D. Goodell, W. Gropp, and R. Thakur, "Hierarchical Collectives in MPICH2," *Proc. 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pp. 325-326, 2009.
[35] J. Zhang, J. Zhai, W. Chen, and W. Zheng, "Process Mapping for MPI Collective Communications," *Proc. 15th Int'l Euro-Par Conf. Parallel Processing (Euro-Par)*, pp. 81-92, 2009.
[36] T. Ma, T. Hérault, G. Bosilca, and J. Dongarra, "Process Distance-Aware Adaptive MPI Collective Communications," *Proc. IEEE Int'l Conf. Cluster Computing (CLUSTER)*, pp. 196-204, 2011.
[37] B. Hendrickson and R. Leland, "The Chaco User's Guide: Version 2.0," Technical Report SAND94–2692, Sandia Nat'l Laboratory, 1994.
[38] UPC Consortium, "UPC Language Specifications, v1.2," Technical Report LBNL-59208, Lawrence Berkeley Nat'l Lab, 2005.
[39] D.H. Bailey, E. Barszcz, L. Dagum, and H. Simon, "NAS Parallel Benchmark Results," Technical Report 94-006, RNR, 1994.
[40] J.C. Hayes, M.L. Norman, R.A. Fiedler, J.O. Bordner, P.S. Li, S.E. Clark, A. ud-Doula, and M.-M. McLow, "Simulating Radiating and Magnetized Flows in Multiple Dimensions with ZEUS-MP," *The Astrophysical J. Supplement*, vol. 165, no. 1, pp. 188-228, 2006.
[41] M. Kneser, "Aufgabe 300," *Jahresber. Deutsch. Math. -Verein 58*, 1955.
[42] A. Kako, T. Ono, T. Hirata, and M.M. Halldorsson, "Approximation Algorithms for the Weighted Independent Set Problem," *Proc. 31st Int'l Workshop Graph-Theoretic Concepts in Computer Science (WG '05)*, pp. 341-350, 2005.

Emmanuel Jeannot received the master's and PhD degrees in computer science in 1996 and 1999, respectively both from Ecole Normale Supérieure de Lyon, at the LIP laboratory. After the PhD degree, he spent one year as a postdoc at the LaBRI laboratory in Bordeaux. He is a research scientist at Institut National de Recherche en Informatique et en Automatique (INRIA) and he has been conducting his research at INRIA Bordeaux Sud-Ouest and at the LaBRI laboratory since September 2009. Before that, he held the same position at INRIA Nancy Grand-Est. From January 2006 to July 2006, he was a visiting researcher at the University of Tennessee, ICL laboratory. From September 1999 to September 2005, he was an assistant professor at the Université Henry Poincaré, Nancy 1. During the period 2000-2009, he did research at the LORIA laboratory. His main research interests include scheduling for heterogeneous environments and grids, data redistribution, algorithms and models for parallel machines, grid computing software, adaptive online compression and programming models.

Guillaume Mercier received the PhD degree in computer science from the University of Bordeaux in 2004. Since 2006, he has been an assistant professor at the Bordeaux Polytechnic Institute. He has been working on parallel programming and MPI for more than 10 years and has been involved in the development of several MPI implementations, especially MPICH2, where he participated in the design and development of the NEMESIS communication layer. His research interests encompass high-performance networks, hierarchical architectures management, parallel programming paradigms, and message passing.

François Tessier received the master's degree in computer science in 2010 from the University of Bordeaux. Since October 2011, he has been working toward the PhD degree at both the University of Bordeaux and Inria. His work deals with process placement on heterogenous architectures.