Process Placement in Multicore Clusters: Algorithmic Issues and Practical Techniques

Emmanuel Jeannot, Guillaume Mercier, François Tessier

Process Placement in Multicore Clusters: Algorithmic Issues and Practical Techniques

Emmanuel Jeannot, Guillaume Mercier, François Tessier

Process Placement in Multicore Clusters:
Algorithmic Issues and Practical Techniques

Emmanuel Jeannot*, Guillaume Mercier†*, François Tessier*

Project-Team Runtime

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 report, 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.

Key-words: Parallel programming, High performance computing, Multicore processing

---

* INRIA Bordeaux Sud-Ouest
† Institut Polytechnique de Bordeaux

Résumé :   Les générations actuelles de grappes de nœuds NUMA possèdent des processeurs multicœurs ou *manycore*. Le programmation efficace de telles architectures est un véritable défi parce que de nombreux détails matériels doivent être pris en considération, en particulier la hiérarchie mémoire. Afin d'améliorer les performances des applications parallèles, une idée séduisante est de diminuer le coût de leurs communications en faisant correspondre leur schéma de communication à l'architecture matérielle sous-jacente. Dans ce rapport de recherche, nous détaillons l'algorithme et les techniques proposés afin d'obtenir ce résultat : d'abord, nous collectons deux informations-clefs, à savoir, le schéma de communication et les détails matériels de l'architecture-cible. Ensuite, nous calculons une permutation des numéros de rang des processus de l'application. Pour finir, ces nouveaux numéros de rang sont utilisés dans les opérations de communication en vue de diminuer les coûts de communication de l'application.

Mots-clés :   Programmation parallèlle, Calcul haute-performance, Programmation multicœur

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 [DBea11].

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 [PRA12]. In this report, 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 report 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) [Mes94]. 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 [HRR$^+$11].

This report 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 report.

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 main standard for programming with parallel processes is MPI. Therefore, in the remainder of this report, 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++ [KK93] 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 report 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 [MTM+09].

We provide now several examples of communication patterns; what is shown is the number of messages exchanged between processes. The rank numbers correspond to that of `MPI_COMM_WORLD`. Figure 1 shows the pattern for the CG kernel of the NAS benchmarks, Figure 2 shows FT and Figure 3 shows LU. For Zeus/MP, Fig. 4 shows the pattern for 64 processes.
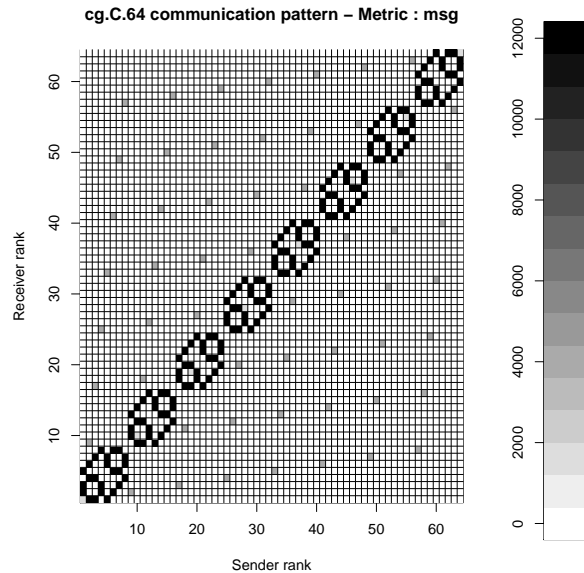

Figure 1: Communication pattern of CG.C.64

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,
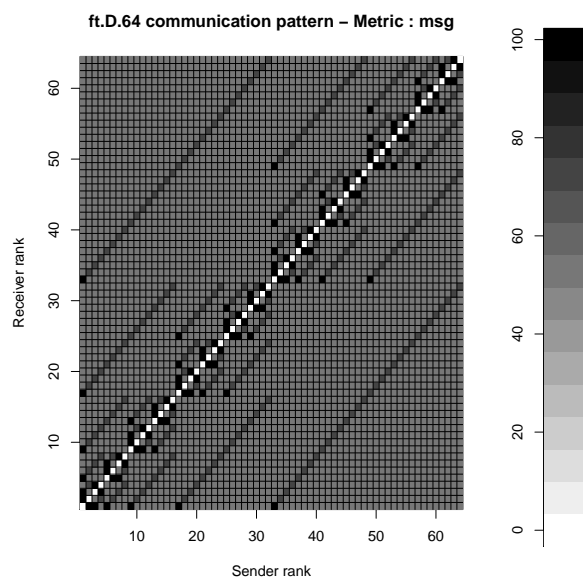
Figure 2: Communication pattern of FT.D.64

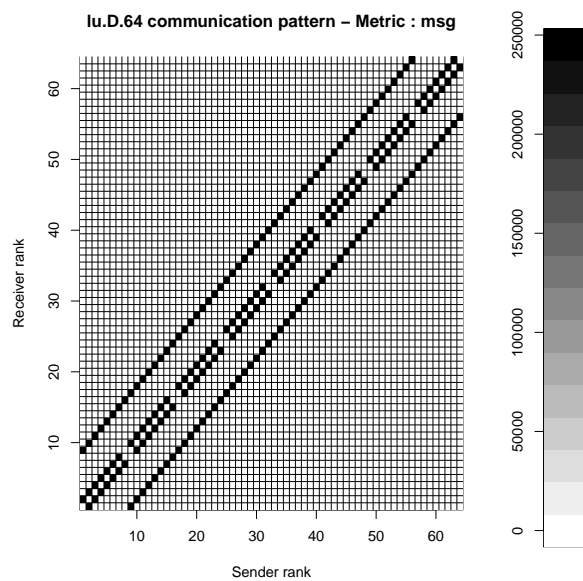Figure 3: Communication pattern of LU.D.64

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
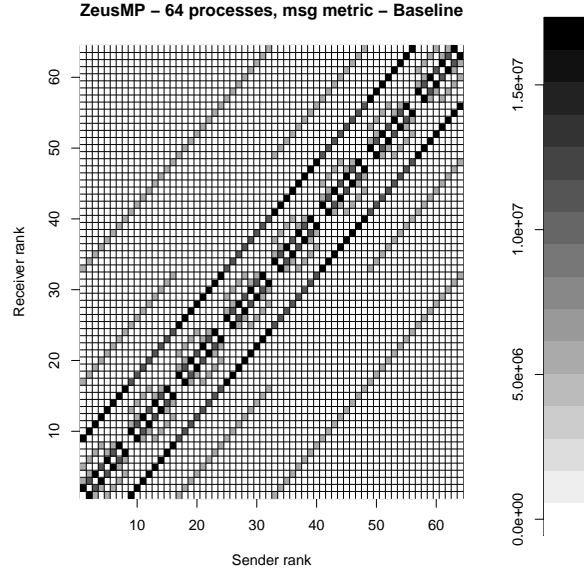
Figure 4: Communication of Zeus/MP 64

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 [D. 07a]) 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.)

---

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

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.

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 [BCOM+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 [MCO09] 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. Enforcing the placement policy means that each MPI process has to be executed

---

[2]The same approach as in [CCH+06].

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 ([MCO09],[RMNP09]). Generally speaking, binding the processes of a parallel application to computing units leads to a decrease of the system noise and improves performance: when the processes are bound to computing units, the application is able to deliver more stable and predictable performance. Indeed, the standard deviation of the overall execution time is decreased, as shown in Table 1. When a process is not bound to a specific computing unit, the operating system scheduler may swap it to another computing unit, leading to cache misses that harm performance. However, as the scheduling of processes is not deterministic, the impact on the performance varies from one run to another. That is why the standard deviation of several runs is lower when binding is enforced.

Number of Iterations	No Binding of Processes	Binding of Processes	Improvement
1000	0.077	0.089	+15%
2000	0.127	0.062	-51%
3000	0.112	0.097	-13%
4000	0.069	0.052	-25%
5000	0.289	0.121	-58%
10000	0.487	0.194	-60%
15000	0.24	0.154	-36%
20000	0.374	0.133	-64%
25000	0.597	0.247	-59%
30000	0.744	0.26	-65%
35000	0.78	0.3	-61%
40000	0.687	0.227	-67%
45000	0.776	0.631	-19%
50000	1.095	0.463	-58%

Table 1: Standard deviation figures for 10 runs of ZEUS-MP/2 CFD application with 64 processes (mhd blast case)

With the *resource binding* technique, 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

---

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

limited (a few dozen code lines, see Appendix A for examples). `MPI_Dist_graph_create` (part of the standard since MPI 2.2 [HRR$^+$11]) 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. [MJ11] 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 report, a *partially distributed* version in which each node reorders only its (local) MPI processes.

In this case, scalability is improved but the intial 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.

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 [Hat98] and an algorithm based on the Kernighan-Lin heuristic [B. 70] 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 ([SB05], [YCM06], or [BGVB11]). InfiniBand fabric is also a suject of studies: [RGB$^+$11] and [IGO12] empirically assess the performance of the interconnection network to provide a usable model of the underlying architecture. [SPK$^+$12] uses the Neighbor Joining Method to detect the physical topology of the underlying InfiniBand network. LibTopoMap [HS11] also considers generic network topologies and relies on ParMETIS [KK95] 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 [J. 02]. 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 [CCH$^+$06] 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 [BAG12], 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 [BAG12] is different from our work. Indeed, [BAG12] 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 [RMNP09] and [MCO09]. 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 [F. 94] partitioner and they are unable to gather the hardware information dynamically at runtime. Also, [RMNP09] 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 [Arg04] and Open MPI [GFB⁺04] 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., [HSD11]). 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 ([Nat], [J. 11]), HP [D. 07b] and IBM (according to [E. 08]).

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, [ZGGT09] uses a hierarchical two-level scheme to make better use of multicore nodes. [ZZCZ09] and [MHBD11] 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 [HL94] and Metis [KK95] (or ParMetis for its parallel version) are examples of such graph-partitioning software. SCOTCH [F. 94] 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++ [KK93], 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 [UPC05]) expose a simple two-level scheme (local and remote) for memory affinity that can be used for mapping processes.

4 The TREEMATCH Algorithm

4.1 Regular version of TREEMATCH

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 [JM10].
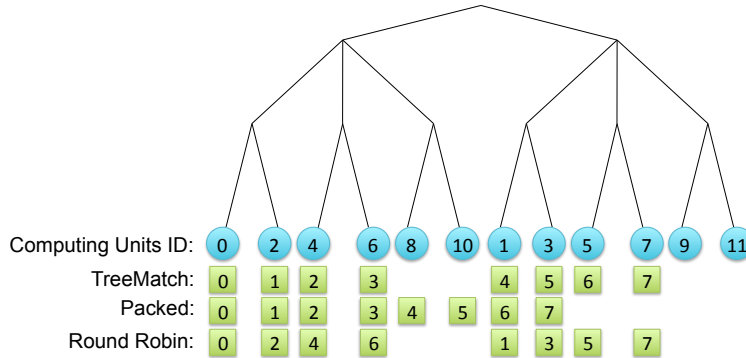

Algorithm 1: The TREEMATCH Algorithm

Input: T // The topology tree
Input: m // The communication matrix
Input: D // The depth of the tree

1 groups$[1..D-1]$=$\emptyset$ // How nodes are grouped on each level
2 **foreach** depth$\leftarrow D-1..1$ **do** // We start from the leaves
3       $p \leftarrow$ order of m
     // Extend the communication matrix if necessary
4   **if** p mod **arity**$(T, depth-1) \neq 0$ **then**
5           $m \leftarrow$ **ExtendComMatrix**$(T,m,$depth$)$

6   groups$[$depth$]\leftarrow$**GroupProcesses**$(T,m,$depth$)$// Group processes by communication affinity
7       $m \leftarrow$ **AggregateComMatrix**$(m,$groups$[$depth$])$ // Aggregate communication of the group of processes
8 MapGroups$(T,$groups$)$ // Process the groups to built the mapping

---

Proc	0	1	2	3	4	5	6	7
0	0	1000	10	1	100	1	1	1
1	1000	0	1000	1	1	100	1	1
2	10	1000	0	1000	1	1	100	1
3	1	1	1000	0	1	1	1	100
4	100	1	1	1	0	1000	10	1
5	1	100	1	1	1000	0	1000	1
6	1	1	100	1	10	1000	0	1000
7	1	1	1	100	1	1	1000	0

(a) Communication Matrix

(b) Topology Tree (squares represent mapped processes using different algorithms)

Figure 5: Input Example of the TREEMATCH Algorithm

For TREEMATCH, we assume that the topology tree is balanced (leaves are all at the same depth) and symmetric (all the nodes of a given depth possess the same arity). Such assumptions are indeed very realistic in the case of a homogeneous parallel machine where all processors, sockets, nodes or cabinets are identical. In order to optimize the communication time of an

application, the TREEMATCH algorithm will map processes to cores depending on the amount of data they exchange.

To describe how the TREEMATCH algorithm works we will run it on the example given in Figure 5. Here, the topology is modeled by a tree of depth 4 with 12 leaves (computing units). The communication pattern between MPI processes is modeled by an 8×8 matrix (hence, we have eight processes). The algorithm processes the tree upward at depth 3. At this depth, we call the arity of the node of the next level k. In our case $k = 2$ and divides the order $p = 8$ of the matrix m. Hence, we directly go to line 6 where the algorithm calls the function GroupProcesses. This information is given by a communication matrix, which can be determined as explained in Section 2.1.

Function GroupProcesses(T,m,depth)

    Input: T //The topology tree
    Input: m // The communication matrix
    Input: depth // current depth
1 $l \leftarrow$ListOfAllPossibleGroups(T,m,depth)
2 $G \leftarrow$GraphOfIncompatibility(l)
3 **return** IndependentSet(G)

This function first builds the list of possible groups of processes. The size of the group is given by the arity k of the node of the tree at the upper level (here 2). For instance, we can group process 0 with processes 1 or 2 up to 7 and process 1 with processes 2 up to 7 and so on. Formally we have $\binom{2}{8} = 28$ possible groups of processes. As we have $p = 8$ processes and we will group them by pairs (k=2), we need to find $p/k = 4$ groups that do not have processes in common. To find these groups, we will build the graph of incompatibilities between the groups (line 2). Two groups are *incompatible* if they share a process (e.g., group (2,5) is incompatible with group (5,7) as process 5 cannot be mapped at two different locations). In this graph of incompatibilities, vertices correspond to the groups and we have an edge between two vertices if the corresponding groups are incompatible. In the literature, such a graph is referred to as the complement of a Kneser Graph [Kne55]. The set of groups we are looking for is thus an *independent set* of this graph.

A valuable property of the complement of the Kneser graph is that since k divides p, any maximal independent set is maximum and of size p/k. Therefore, any greedy algorithm always finds an independent set of the required size. However, all grouping of processes (i.e., independent sets) are not of equal quality. They depend on the values of the matrix. In our example, grouping process 0 with process 5 is not a good idea as they exchange only one piece of data and if we group them we will have a lot of remaining communication to perform at the next level of the topology. To account for this, we valuate the graph with the amount of communication reduced thanks to this group. For instance, based on the matrix m, the sum of communication of process 0 is 1114 and of process 1 is 2104 for a total of 3218. If we group them together, we will reduce the communication volume by 2000. Hence, the valuation of the vertex corresponding to group (0,1) is 3218-2000=1218. The smaller the value, the better the grouping.

Unfortunately, finding such an *independent set* of minimum weight is NP-Hard and inapproximable at a constant ratio [KOHH05]. Therefore, we use heuristics to find a "good" independent set:

- **smallest values first**: we rank vertices by smallest values first and we build a maximal independent set greedily, starting with the vertices with smallest values.

- **largest values last**: we rank vertices by smallest values first and we build a maximal independent set such that the largest index of the selected vertices is minimized.

- **largest weighted degrees first**: we rank vertices by their decreasing weighted degrees (the average weight of their neighbors) and we build a maximal independent set greedily, starting with the vertices with largest weighted degrees [KOHH05].

In our implementation we start with the first method and try to improve the solution by applying the last two. We can use a user-defined threshold value to disable the weighted degrees technique when the number of possible groups is too large. In our case, regardless of the heuristic we use, we find the independent set of minimum weight, which is $\{(0,1),(2,3),(4,5),(6,7)\}$. This list is affected to the array group[3] in line 6 of the TREEMATCH algorithm. This means that, for instance, process 0 and process 1 will be put on leaves sharing the same predecessor.

Virt. Proc	0	1	2	3
0	0	1012	202	4
1	1012	0	4	202
2	202	4	0	1012
3	4	202	1012	0

(a) Aggregated matrix (depth 2)

Virt. Proc	0	1	2	3	4	5
0	0	1012	202	4	0	0
1	1012	0	4	202	0	0
2	202	4	0	1012	0	0
3	4	202	1012	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0

(b) Extended matrix

Virt. Proc	0	1
0	0	412
1	412	

(c) Aggregated matrix (depth 1)

Figure 6: Evolution of the communication matrix at different steps of the algorithm

Then, we build the groups at depth 2, but we need to aggregate the matrix m with the remaining communication beforehand. The aggregated matrix is computed in the `AggregateComMatrix` Function.

Function AggregateComMatrix(m,g)

    Input: m // The communication matrix
    Input: g // list of groups of (virtual) processes to merge
1 $n \leftarrow$ NbGroups(g)
2   **for** $i \leftarrow 0..(n-1)$ **do**
3   **for** $j \leftarrow 0..(n-1)$ **do**
4   **if** $i = j$ **then**
5 $r[i,j] \leftarrow 0$
6   **else**
7 $r[i,j] \leftarrow \sum_{i_1 \in g[i]} \sum_{j_1 \in g[j]} m[i_1,j_1]$

8   **return** r

The goal is to compute the remaining communication between each group of processes. For instance, between the first group $(0,1)$ and the second group $(2,3)$ the amount of communication is 1012 and is put in $r[0,1]$ (see Figure 6(a)). The matrix r is of size 4×4 (we have four groups) and is returned to be affected to m (line 7 of the TREEMATCH algorithm). Now, the matrix m corresponds to the communication pattern between groups of processes (called virtual processes)

built during this step. The goal of the remaining steps of the algorithm is to group these virtual processes up to the root of the tree.

Function ExtendComMatrix(T,m,depth)

Input: T //The topology tree
Input: m // The communication matrix
Input: depth // current depth
1 $p \leftarrow$ order of m
2 $k \leftarrow$ arity(T,depth+1)
3 **return** AddEmptyLinesAndCol(m,k,p)

The algorithm then loops and decrements depth to 2. Here, the arity at depth 1 is 3 and does not divide the order of m (4). Hence, we add two artificial groups that do not communicate with any other groups. This means that we add two lines and two columns full of zeroes to the matrix m. The new matrix is depicted in Figure 6(b). The goal of this step is to allow more flexibility in the mapping, thus yielding a more efficient mapping.

Once this step is performed, we can group the virtual processes (group of processes built in the previous step). Here the graph modeling and the independent set heuristics lead to the following mapping: {(0,1,4),(2,3,5)}. Then we aggregate the remaining communication to obtain a 2×2 matrix (see Figure 6(c)). During the next loop (depth=1), we have only one possibility to group the virtual processes: {(0,1)}, which is affected to group[1].

The algorithm then goes to line 8. The goal of this step is to map the processes to the resources. To perform this task, we use the groups array, which describes a hierarchy of groups of processes. A traversal of this hierarchy gives the process mapping. For instance, virtual process 0 (resp. 1) of group[1] is mapped on the left (resp. right) part of the tree. When a group corresponds to an artificial group, no processes will be mapped to the corresponding sub-tree. At the end, processes 0 to 7 are mapped to leaves (computing units) 0,2,4,6,1,3,5,7, respectively (see bottom of Figure 5(b)). This mapping is optimal. The algorithm provides an optimal solution if the communication matrix corresponds to a hierarchical communication pattern (processes can be arranged in a tree, and the closer they are in this tree the more they communicate), that can be mapped to the topology tree (such as the matrix of Figure 5(a)). In this case, optimal groups of (virtual) processes are automatically found by the independent set heuristic, as the corresponding weights of these groups are the smallest among all the groups. Moreover, thanks to the creation of artificial groups (line 5), we avoid the *Packed* mapping 0,2,4,6,8,1,3, which is worse as processes 4 and 5 communicate a lot with processes 6 and 7 and hence must be mapped to the same sub-tree. On the same figure, we can observe that the *Round Robin* mapping, which maps process i on computing unit i, leads also to a suboptimal result.

The main cost of the algorithm is in the function `GroupProcesses`. 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 $\binom{p}{k} = O(p^k)$.

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 B 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 some cases, flattening the network does not significantly hinder the performance: in Figure 7, we present the timings of several NAS kernels on 128 computing units (16 nodes) for both classes C and D. Two cases are studied: in the first, all the nodes are on the same switch while in the second, computing nodes are dispatched evenly on two switches, incurring more costly communications. The solution computed by TREEMATCH is the same in both cases. Results show that, nevertheless, the impact on the performance for these two cases is small. Moreover, as explained in the State-of-the-Art Section, complex network topologies are addressed by other works in a complementary fashion.
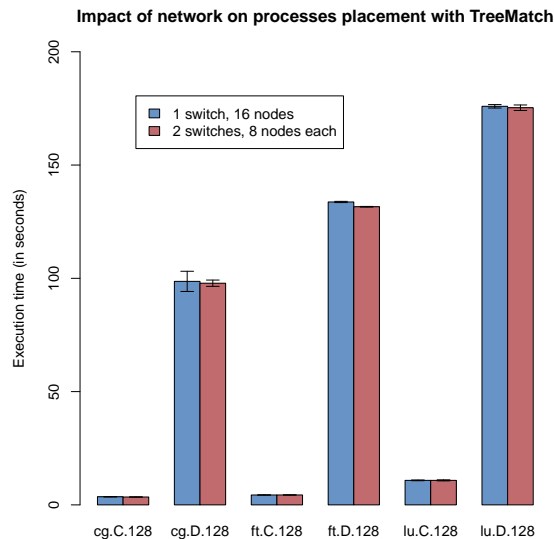

Figure 7: Network fragmentation

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`. 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 article, 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.

4.2   Running Time Optimization of TREEMATCH

4.2.1   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:

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

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 $p \leq 500\,000$. 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.

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

---

[4]most of these computations can be checked using Matlab

k	p^*	d^*		k	p^*	d^*		k	p^*	d^*
4	8	2		49	58	7		90	94	45
6	10	3		50	54	25		91	100	13
8	12	4		51	56	17		92	96	46
9	14	3		52	56	26		93	98	31
10	14	5		54	58	27		94	98	47
12	16	6		55	62	11		95	102	19
14	18	7		56	60	28		96	100	48
15	20	5		57	62	19		98	102	49
16	20	8		58	62	29		99	104	33
18	22	9		60	64	30		100	104	50
20	24	10		62	66	31		102	106	51
21	26	7		63	68	21		104	108	52
22	26	11		64	68	32		105	110	35
24	28	12		65	72	13		106	110	53
25	32	5		66	70	33		108	112	54
26	30	13		68	72	34		110	114	55
27	32	9		69	74	23		111	116	37
28	32	14		70	74	35		112	116	56
30	34	15		72	76	36		114	118	57
32	36	16		74	78	37		115	122	23
33	38	11		75	80	25		116	120	58
34	38	17		76	80	38		117	122	39
35	42	7		77	86	11		118	122	59
36	40	18		78	82	39		119	128	17
38	42	19		80	84	40		120	124	60
39	44	13		81	86	27		122	126	61
40	44	20		82	86	41		123	128	41
42	46	21		84	88	42		124	128	62
44	48	22		85	92	17		125	132	25
45	50	15		86	90	43		126	130	63
46	50	23		87	92	29		128	132	64
48	52	24		88	92	44				

Table 2: Table of optimal tree division. Given a node of arity k it tells above which number of processors p^* it is useful to divide this node into d^* nodes of arity k/d^*. It is based on the fact that $\forall p \geq p^*, \binom{p}{k} > d^*\binom{p}{k/d^*}$.

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, 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.

4.2.2   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. 8 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 [BBDS94]. 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 [J. 06]. 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 (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.

To represent architectures, Scotch proposes several formats. When dealing with a hierarchical, tree-structured topology, Scotch uses the *tleaf* built-in definition. A tleaf file is a single line file with the following syntax: **tleaf** n a_0 v_0 $\ldots a_{n-1}$ v_{n-1}

There are $n+1$ levels in the tree (numbered from 0 to n). The leaves are at the level n. a_i ($0 \leq i \leq n-1$) is the arity of the ith level and v_i ($0 \leq i \leq n-1$) is the traversal cost between level i and $i+1$.

An important issue encountered with Scotch arises from this *tleaf* representation with an edge-weighted tree of the target architecture. 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).

In our 128-computing unit experiments, we used two tleaf structures:

- `tleaf 4 16 4 2 3 2 2 2 1` for the *scotch* case and,

- `tleaf 4 16 500 2 100 2 50 2 10` for the *scotch_w* case.

Each tleaf has the same structure: five levels with arity of intermediate nodes being 16, 2, 2 and 2. Only costs between the levels change. However, this is mainly a change in orders of magnitude. Nevertheless, the scotch_w tleaf leads to worse results than the *scotch* one for the ZEUS/MP experiments as shown in Fig. 14.

---

[5] This policy is typically enforced by batch schedulers when reserving nodes in a cluster for MPI applications.
[6] Figure 5 depicts the difference between RR and *Packed* policies.

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 16384 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.
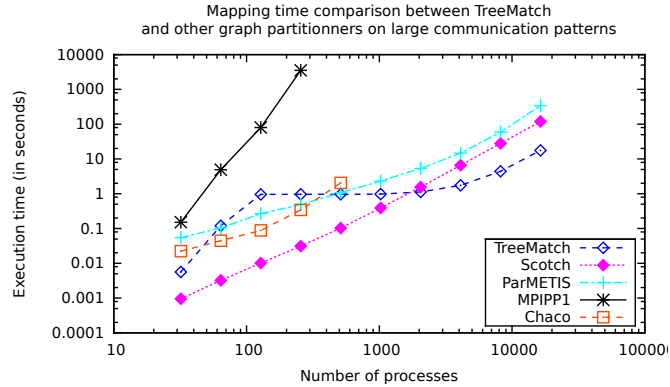
Figure 8: Average mapping computation time comparison for various placement methods

Results are depicted in Figure 8. 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 3550 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 4.2.2) and the slope of its curve flattens. For size 2048 and more, TREEMATCH is the fastest strategy. For size 16384, TREEMATCH is seven times faster than Scotch and 20 times faster than ParMETIS. This demonstrates that TREEMATCH scales better than the other methods.

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.

Figure 9 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 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.
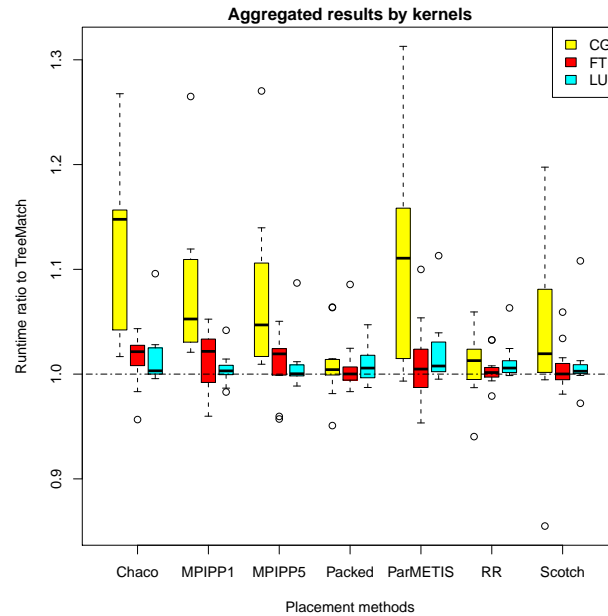
Figure 9: Average execution time ratio between TREEMATCH and other placement methods for the NAS benchmarks. Results projected by kernels (LU, CG and FT). Metric Avg excluded.

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.

We also provide projections for other parameters: the various metrics (Figure 10), the various classes (Figures 12 and 11) and the various process counts considered (Figure 13). In Figure 10, we show the ratios for the three metrics: the amount of exchanged data (*Size*), the number of messages (*Msg*), and the average size of exchanged messages (*Avg*). We see that except for the *Avg* metric for *Packed* and *RR* all the medians are above 1. For the *Size* and *Msg* metrics, almost 75% of the ratios are above one. Gains of more than 10% in execution times are commonplace while we lose more than 10% only in marginal cases.

_____

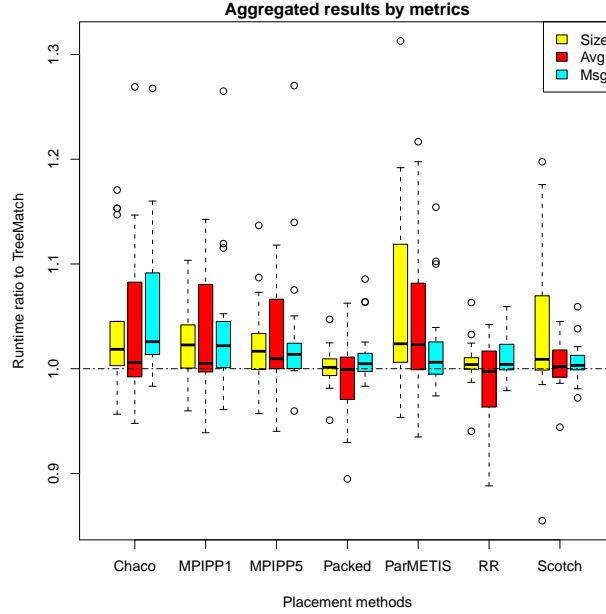[7]See http://en.wikipedia.org/wiki/Box_plot for more details.

Figure 10: Average execution time ratio between TREEMATCH and other placement methods for the NAS benchmarks. Results projected by metrics (number of messages, data size or average message size)

The worst results are obtained when RR and *Packed* policies are compared to TREEMATCH using the *Avg* metric. The actual execution times are in general higher for this metric than for the *Size* and *Msg* ones. Indeed, RR and *Packed* methods do not depend on any metric as they do not use the communication matrix but only the computing units numbering. Hence, it is generally better not to use the average message size metric for computing the process placement. For this reason, we have excluded this metric from Figures 11, 12 and 13.

In Figure 11, we see that the ratio over the other metrics decreases when we increase the class (i.e., the problem size). Indeed, the tested kernels are compute-intensive ones, meaning that when increasing the input size, the computation time grows faster than the communication time. As the process placement helps in improving the communication time, the gain is in proportion less for large inputs than for small inputs. However, if we display the difference between TREEMATCH and the other methods we can see that the gain (in terms of difference) is increasing along with the input size, as shown in Figure 12.

In Figure 13, we see what happens when the number of processes increases. The main result from this figure is that the discrepancy of the ratios tends to increase along with the number of processes. Actually, most of the cases with a ratio greater than one tend to keep a ratio greater than one when we change the number of processes. One of the explanations for the distance to 1.0 increasing with the number of processes is that the computation to communication ratio decreases and effects due to communication are therefore more visible with higher numbers of processes.

Other noticeable results are that MPIPP1 is worse than MPIPP5 whilst Chaco is the worst method of all. Last, ParMETIS does not guarantee that the computed partitions are of equal sizes because it is not designed to produce such partitions. This would explain the poor results achieved with this partitioner.
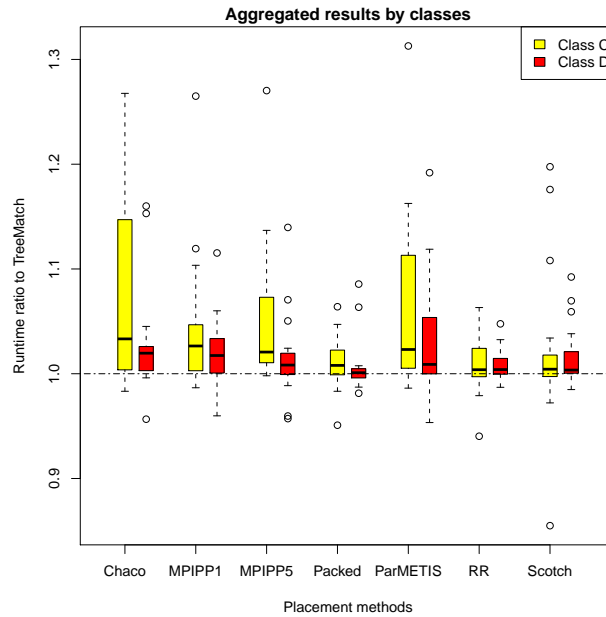
Figure 11: Average execution time ratio between TREEMATCH and other placement methods for the NAS benchmarks. Results projected by Class (C: average problem size, D: large problem size). Metric Avg excluded

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

Figures 14(a), 14(b) and 14(c) 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 % (285.62 s vs 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. 14(c)). TREEMATCH does not suffer from this drawback as it relies only on structural properties of the topology. Moreover, as shown in Fig. 16 of Appendix B, 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 vs. distributed mapping

Figure 5.2.4 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
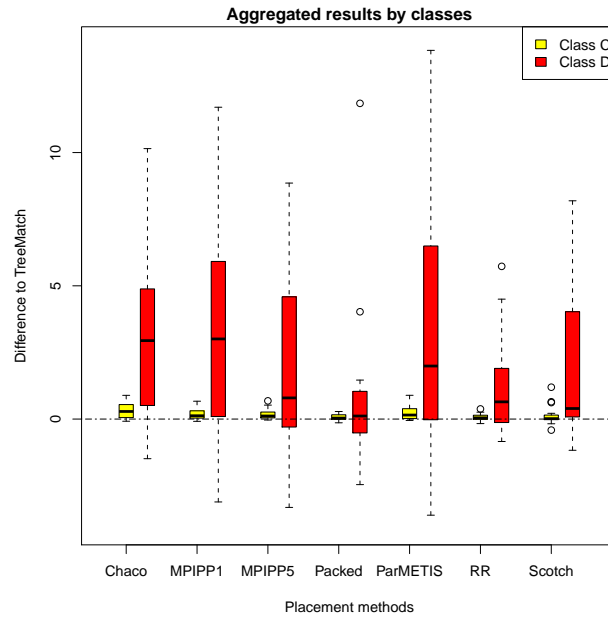
Figure 12: Average execution time difference between TREEMATCH and other placement methods for the NAS benchmark. Results projected by Class (C: average problem size, D: large problem size). Metric Avg excluded.
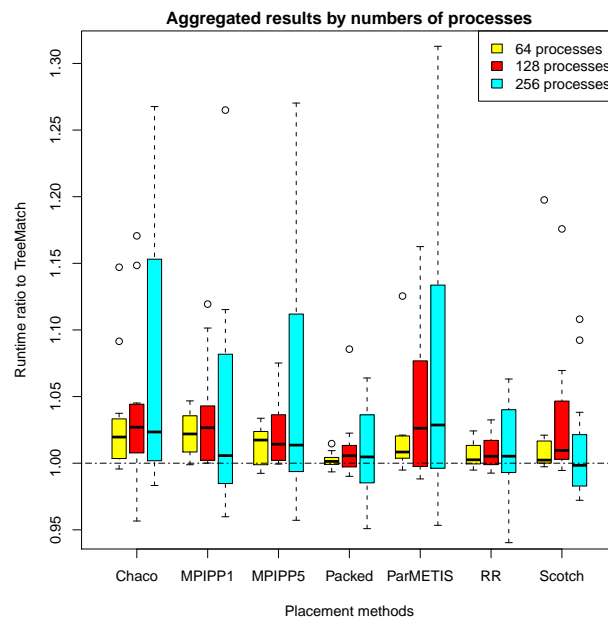

Figure 13: Average execution time ratio between TREEMATCH and other placement methods for the NAS benchmarks. Results projected by number of processes (64, 128, 256). Metric Avg excluded.
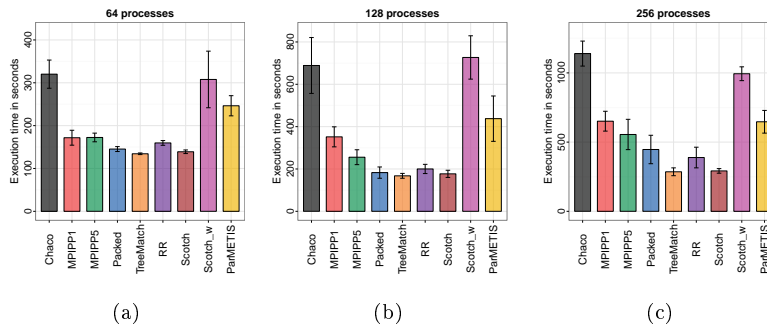
Figure 14: Average execution time of ZEUS-MP/2 on several numbers of processes (average of 10 runs, *msg* metric, 3000 iterations)
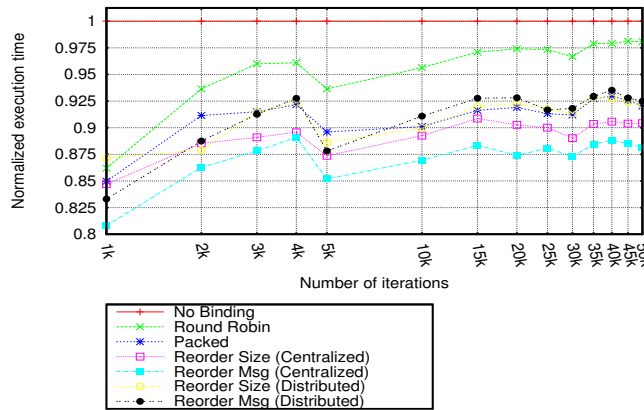

Figure 15: ZEUS-MP/2 (mhd blast case, 64 processes, MVAPICH2): influence of the placement policy on performance

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% (67 s execution time for RR vs. 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 report, 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% 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

[Arg04]   Argonne National Laboratory. MPICH2. http://www.mcs.anl.gov/mpi/2004.

[B. 70]   B. W. Kernighan and S. Lin . An Efficient Heuristic Procedure for Partitioning Graphs. *Bell System Technical Journal*, 49(2):291–307, February 1970.

[BAG12]   B. Brandfass, T. Alrutz, and T. Gerhold. Rank Reordering for MPI Communication Optimization. *Computer & Fluids*, January 2012.

[BBDS94]   D. H. Bailey, E. Barszcz, L. Dagum, and H.D. Simon. NAS Parallel Benchmark Results. Technical Report 94-006, RNR, 1994.

[BCOM+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. In *Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP2010)*, Pisa, Italia, February 2010. IEEE Computer Society Press.

[BGVB11]   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*, 26(3-4):247–256, 2011.

[CCH+06]   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. In G. K. Egan and Y. Muraoka, editors, *ICS*, pages 353–360. ACM, 2006.

[D. 07a]   D. Buntinas, G. Mercier and W. Gropp. Implementation and Evaluation of Shared-Memory Communication and Synchronization Operations in MPICH2 using the Nemesis Communication Subsystem. *Parallel Computing, Selected Papers from EuroPVM/MPI 2006*, 33(9):634–644, September 2007.

[D. 07b]   D. Solt. A Profile Based Approach for Topology Aware MPI Rank Placement, 2007. `http://www.tlc2.uh.edu/hpcc07/Schedule/speakers/hpcc_hp-mpi_solt.ppt`.

[DBea11]   J. Dongarra, P. Beckman, and et al. The International Exascale Software Roadmap. *International Journal of High Performance Computer Applications*, 25(1), 2011.

[E. 08]   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, 2008. `http://www.faqs.org/patents/app/20080288957`.

[F. 94]   F. Pellegrini. Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs. In *Proceedings of SHPCC'94, Knoxville*, pages 486–493. IEEE, may 1994.

[GFB⁺04]   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. In *Proceedings of the 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[Hat98]   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.

[HL94]   B. Hendrickson and R. Leland. The Chaco User's Guide: Version 2.0. Technical Report SAND94–2692, Sandia National Laboratory, 1994.

[HRR⁺11]   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*, 23(4):293–310, 2011.

[HS11]   T. Hoefler and M. Snir. Generic Topology Mapping Strategies for Large-Scale Parallel Architectures. In D. K. Lowenthal, B. R. de Supinski, and S. A. McKee, editors, *ICS*, pages 75–84. ACM, 2011.

[HSD11]   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.

[IGO12]   S. Ito, K. Goto, and K. Ono. Automatically Optimized Core Mapping to Subdomains of Domain Decomposition Method on Multicore Parallel Environments. *Computer & Fluids*, April 2012.

[J. 02]   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.

[J. 06]   J. C. Hayes and M. L. Norman and R. A. Fiedler and J. O. Bordner and P. S.
             Li and S. E. Clark and A. ud-Doula and M-M. McLow. Simulating Radiating
             and Magnetized Flows in Multiple Dimensions with ZEUS-MP. *The Astrophysical
             Journal Supplement* , 165(1):188–228, 2006.

[J. 11]   J. L. Whitt, G. Brook and M. Fahey. Cray MPT: MPI on the Cray XT, 2011.
             `http://www.nccs.gov/wp-content/uploads/2011/03/MPT-OLCF11.pdf`.

[JM10]   E. Jeannot and G. Mercier. Near-Optimal Placement of MPI processes on Hierar-
             chical NUMA Architectures. In Pasqua D'Ambra, Mario Rosario Guarracino, and
             Domenico Talia, editors, *Euro-Par 2010 - Parallel Processing, 16th International
             Euro-Par Conference*, volume 6272 of *Lecture Notes on Computer Science*, pages
             199–210, Ischia Italie, SEPT 2010. Springer.

[KK93]   L.V Kale and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented
             System Based on C++. In *Proceedings of Object-Oriented Programming, Systems,
             Languages and Applications (OOPSLA) 93*, pages 91–108. ACM Press, September
             1993.

[KK95]   G. Karypis and V. Kumar. METIS - Unstructured Graph Partitioning and Sparse
             Matrix Ordering System, Version 2.0. Technical report, 1995.

[Kne55]   M. Kneser. Aufgabe 300. *Jahresber. Deutsch. Math. -Verein 58*, 1955.

[KOHH05]   A. Kako, T. Ono, T. Hirata, and M. M. Halldorsson. Approximation Algorithms
             for the Weighted Independent Set Problem. In *LNCS*, number 3787, pages 341–350.
             SPRINGER-VERLAG, 2005.

[MCO09]   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.

[Mes94]   Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*,
             March 1994.

[MHBD11]   T. Ma, T. Hérault, G. Bosilca, and J. Dongarra. Process Distance-Aware Adap-
             tive MPI Collective Communications. In *2011 IEEE International Conference on
             Cluster Computing (CLUSTER)*, pages 196–204. IEEE, 2011.

[MJ11]   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.

[MTM+09]   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. In *Pro-
             ceedings of 23rd IEEE International Parallel and Distributed Processing Symposium
             (IPDPS'09)*, Rome, Italy, May 2009. IEEE Computer Society Press.

[Nat]   National Institute for Computational Sciences. MPI Tips on Cray XT5. `http:
             //www.nics.tennessee.edu/user-support/mpi-tips-for-cray-xt5`.

[PRA12]   PRACE. The scientific case for high performance computing in Europe. report,
             2012.   P. 147, `http://www.prace-ri.eu/IMG/pdf/prace_-_the_scientific_
             case_-_executive_s.pdf`.

[RGB⁺11]   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*, volume 6960 of *Lecture Notes in Computer Science*, pages 50–60. Springer, 2011.

[RMNP09]   E. Rodrigues, F. Madruga, P. Navaux, and J. Panetta. Multicore Aware Process Mapping and its Impact on Communication Overhead of Parallel Applications. In *Proceedings of the IEEE Symp. on Comp. and Comm.*, pages 811–817, July 2009.

[SB05]   B. E. Smith and B. Bode. Performance Effects of Node Mappings on the IBM BlueGene/L Machine. In J. C. Cunha and P. D. Medeiros, editors, *Euro-Par*, volume 3648 of *Lecture Notes in Computer Science*, pages 1005–1013. Springer, 2005.

[SPK⁺12]   H. Subramoni, S. Potluri, K. Kandalla, B. Barth, J. Vienne, J. Keasler, K. Tomko, K. Schulz, A. Moody, and D.K. Panda. Design of a Scalable Infiniband Topology Service to Enable Network-Topology-Aware Placement of Processes . In *Proceedings of the 2012 ACM/IEEE conference on Supercomputing (CDROM)*, page 12, Salt Lake City, Utah, United States, 2012. IEEE Computer Society.

[UPC05]   UPC Consortium. UPC Language Specifications, v1.2. In *Lawrence Berkeley National Lab Tech Report LBNL-59208*, 2005.

[YCM06]   H. Yu, I-H. Chung, and J. E. Moreira. Blue Gene System Software - Topology Mapping for Blue Gene/L Supercomputer. In *SC*, page 116. ACM Press, 2006.

[ZGGT09]   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.

[ZZCZ09]   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.

A   Modifications to Legacy MPI Source Codes

We modified legacy MPI applications to issue a call to the MPI 2.2 `MPI_Dist_graph_create` routine in order to create a new communicator (`comm_to_use`) in which the processes ranks are reordered. This call is made (and the reordering computed) just after the initialization step and before any application data are loaded into the MPI processes, otherwise data movements are necessary. Then, all relevant occurrences of `MPI_COMM_WORLD` are replaced by `comm_to_use` in the rest of the code. We provide as a commodity a routine (`read_pattern`) that gets the pattern from a trace file generated by a previous run of the target application[8].

A.1   An example of C application: the IS NAS kernel

In the case of the IS NAS kernel, two new program arguments are used: the first one is the name of the pattern information file while the second argument is the metric to be used (`size`, `msg` or `avg`). Also, fourteen occurrences of `MPI_COMM_WORLD` have been replaced by `comm_to_use` in the rest of the code (file `is.c`).
Here is the complete modified code:

```
/*  Initialize MPI */
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &my_rank );
    MPI_Comm_size( MPI_COMM_WORLD, &comm_size );

    if(argc > 1){
      int  reorder = 1;

      if (my_rank == 0){
          int *sources      = NULL;
          int *degrees      = NULL;
          int *destinations = NULL;
          int *weights      = NULL;

          read_pattern(pattern_file,comm_size,
                       &destinations,&weights,
                       &sources,&degrees,metric_to_use);

          MPI_Dist_graph_create(MPI_COMM_WORLD,comm_size,
                                sources,degrees,
                                destinations,weights,
                                MPI_INFO_NULL,
                                reorder, &comm_topo);
          free(sources);
          free(degrees);
          free(destinations);
          free(weights);
        } else {
          MPI_Dist_graph_create(MPI_COMM_WORLD,0,
                                NULL,NULL,
                                NULL,NULL,
                                MPI_INFO_NULL,
                                reorder, &comm_topo);
      }

      if ( comm_topo != MPI_COMM_NULL )
          comm_to_use = comm_topo;
       else
          comm_to_use = MPI_COMM_WORLD;

      MPI_Comm_rank( comm_to_use, &my_rank);
    }
```

---

[8]The users are supposed to provide such a pattern and feed it directly to the `MPI_Dist_graph_create` function through its `destinations`,`weights`, `sources` and `degrees` parameters.

A.2   An example of Fortran application: ZEUS-MP/2

The ZEUS-MP/2 source code modifications follow the same scheme as the previous example:

1. A new variable `comm_to_use` has been introduced in the file `mod_files.F`.

2. The file `configure.F` has been modified to make the call to `MPI_Dist_graph_create` (as shown below).

3. MPI_COMM_WORLD occurrences have been replaced by `comm_to_use` in the following source files: `mstart.F` (seven occurrences), `rshock.F` (two occurrences), `setup.F` (14 occurrences), `marshak.F` (one occurrence), `restart.F` (two occurrences), `fftwplan.c` (two occurrences) and `fftw_ps.c` (two occurrences).

Here is an excerpt of the `configure.F` file demonstrating the use of the `MPI_Dist_graph_create` function:

```
#ifdef MPI_USED
c
c Reordering modifs
c
      character name*64, mode*1
      integer switch_comm
      integer num_degrees, numargs, newmode
      integer, allocatable, dimension(:) :: sources
      integer, allocatable, dimension(:) :: degrees
      integer, allocatable, dimension(:) :: destinations
      integer, allocatable, dimension(:) :: weights
#endif
c
c-----------------------------------------------------------
c      If parallel execution, start up MPI
c-----------------------------------------------------------
c
#ifdef MPI_USED
      call MPI_INIT( ierr )
      call MPI_COMM_RANK( MPI_COMM_WORLD, myid_w  , ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs_w, ierr )
c
c Reordering Modifs
c
      reorder = .true.
      switch_comm = 0

      numargs = iargc()
      if (numargs .eq. 2) then
        if(myid_w .eq. 0) then
           allocate(sources(nprocs_w))
           allocate(degrees(nprocs_w))
           num_degrees = 0
           call getarg(1, pattern_file)
           call read_pattern_fortran_get_degrees(num_degrees,
     >            pattern_file)

           allocate(destinations(num_degrees))
           allocate(weights(num_degrees))

           call getarg(2, metric_to_use)
           read( metric_to_use, '(i10)' ) newmode
           call read_pattern_fortran(nprocs_w, num_degrees,
     >            destinations(1), weights(1), sources(1),
     >            degrees(1),newmode,pattern_file)

           call MPI_DIST_GRAPH_CREATE(MPI_COMM_WORLD,
     >            nprocs_w,sources,degrees,destinations,
     >            weights, MPI_INFO_NULL,
     >            reorder,comm_topo,ierr)

           deallocate(sources,stat = ierr)
```

```
            deallocate(degrees,stat = ierr)
            deallocate(destinations,stat = ierr)
            deallocate(weights,stat = ierr )
         else
            call MPI_DIST_GRAPH_CREATE(MPI_COMM_WORLD, 0,
     >               0, 0, 0, 0,
     >               MPI_INFO_NULL,
     >               reorder,comm_topo,ierr)
         endif
         switch_comm = 1
      endif

      if(switch_comm .eq. 1) then
         comm_to_use = comm_topo
      else
         comm_to_use = MPI_COMM_WORLD
      endif

      call MPI_COMM_RANK(comm_to_use, myid_w, ierr)
      reorder = .false.
#else
      myid_w = 0
      myid   = 0
#endif /* MPI_USED */
```

B   Bandwidth Measurements Between Computing Units

I In Figure 16 we present timings measuring the bandwidth obtained for transmitting a given
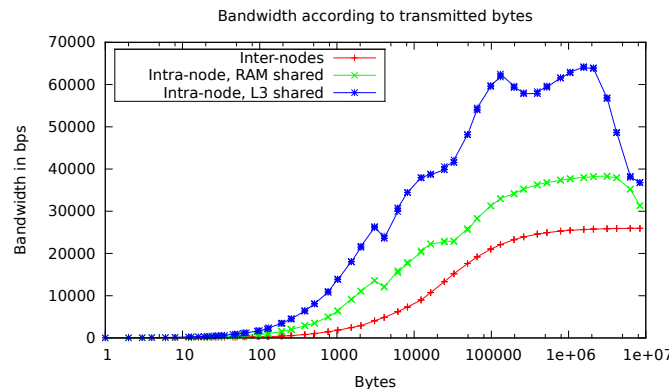
Figure 16: ZEUS-MP/2 (metric: msg, 256 processes)

number of bytes on PLAFRIM using the NetPIPE[9] tool. Three cases are shown, depending on whether the sender and the receiver share the same L3 cache, share a memory bank, or are on different nodes. We see that the bandwidth is always the highest for the first case and always the lowest for the last cases. Moreover, we see that the performance is not linear (and not affine) in any cases. Overall, these experiments strengthen the structural models exploited by TREEMATCH compared to the quantitative approach followed by other tools such as Scotch.

---

[9] http://www.scl.ameslab.gov/netpipe/

Contents