Near-Optimal Placement of MPI processes on Hierarchical NUMA Architectures

Emmanuel Jeannot, Guillaume Mercier

Near-Optimal Placement of MPI processes on Hierarchical NUMA Architectures

Emmanuel Jeannot[2,1] and Guillaume Mercier[3,2,1]

[1] LaBRI
[2] INRIA Bordeaux Sud-Ouest
[3] Institut Polytechnique de Bordeaux
{Emmanuel.Jeannot|Guillaume.Mercier}@labri.fr

Abstract. MPI process placement can play a deterministic role concerning the application performance. This is especially true with nowadays architecture (heterogenous, multicore with different level of caches, etc.). In this paper, we will describe a novel algorithm called TreeMatch that maps processes to resources in order to reduce the communication cost of the whole application. We have implemented this algorithm and will discuss its performance using simulation and on the NAS benchmarks.

1 Introduction

The landscape of parallel computing has undergone tremendous changes since the introduction of multicore architectures. Multicore machines feature hardware characteristics that are a novelty, especially when compared to cluster-based architectures. Indeed, the amount of cores available within each system is much higher and the memory hierarchy becomes much more complex than previously. Thus, the communication performance can dramatically change according to the processes location within the system since the closer the data is located from the process, the faster the access shall be. This is know as the Non-Uniform Memory Access (NUMA) effect and can be commonly experienced in modern computers. As the core amount in a node is expected to grow sharply in the near future, all these changes have to be taken into account in order to exploit such architectures at their full potential. However, there is a gap between the hardware and the software. Indeed, as far as programming is concerned, the change is less drastic since users still rely on standards such as MPI or OpenMP. *Hybrid* programming (that is, mixing both message-passing and shared memory paradigms) is one of the keys to obtain the best performance from hierarchical multicore machines. This implies new programming practices that users should follow and apply. However, legacy MPI applications can already take advantage of the computing power offered by such complex architectures. One way of achieving this goal is to match an application's communication pattern to the underlying hardware. That is, the processes that communicate the most would be bound on cores that share the most levels in the memory hierarchy (*e.g.* caches). The idea is therefore to build a correspondence between the list of MPI process ranks and the list of core numbers in the machine. Thus, the placement of MPI processes relies totally on the matching that is computed by a relevant algorithm.

Then, the issue is to use an algorithm that yields a satisfactory solution to our problem. In this paper, we introduce a new algorithm, called TreeMatch that efficiently computes a solution to our problem by taking into account the specificities of the underlying hardware. The rest of this paper is organized as follows: in section 2 we will describe some related works. Section 3 exposes the problem and show how it can be modeled while section 4 describes our TreeMatch algorithm. Both theoretical and empirical results are detailed in section 5. At last, section 6 concludes this paper.

2 Related work

Concerning process placement a pioneer work is provided by Kruskal and Snir in [9] where the problem is modeled by a multicommodity flow. The MPIPP [2] framework takes into consideration a multicluster context and strives to dispatch the various MPI processes on the different clusters used by an application. Graph theory algorithms are widely used in order to determine the matching (a list of *(MPI process rank,core number)* couples). For instance, several vendor MPI implementations, such as the ones provided by Hewlett-Packard [3][4] or by IBM (according to [4]) make use of such mechanism. [6] also formalizes the problem with graphs. In these cases, however, the algorithm computing the final mapping is MPI implementation-specific and does not take into account the complexity of the hierarchy encountered in multicore NUMA machines nor their topologies. In a previous paper [10], we used a graph partitioning algorithm called Scotch [5] to perform the task of mapping computation. However, Scotch is able to work on any type of graphs, not just trees as in our case. Making some hypothesis on the graph structure can lead to improvements and that is why we have developed a new algorithm, called TreeMatch, tailored to fit exactly our specific needs.

3 Problem Modeling

In this section, we describe how we modeled the problem. We first describe how we gather information about the underlying hardware. Then we explain the method used in order to retrieve information about an application's communication pattern.

3.1 Hardware Architecture

The first step for determining a relevant process placement consists of gathering information about the underlying hardware. Retrieving the information about the memory hierarchy in a portable way is not a trivial task. Indeed, no tool is able to provide information about the various caches levels (such as their respective sizes and which cores can access them) on a wide spectrum of systems. To this end, we participated in the development of

[4] HP-MPI has since been replaced by Platform MPI

a specific software tool that fulfills this goal: Hardware Locality or HWLOC [1]. Thanks to HWLOC, we can get all the needed information about the architecture, that is, the number of NUMA nodes, sockets and cores as well as the information about the memory hierarchy. In previous works [2,10] the architecture is modeled by a topology matrix. Entries in this matrix correspond to the communication speed between two cores and take into account the number elements of the memory hierarchy shared between cores. However, such a representation induces a side effect as it "flattens" the view of the hardware's structure. The result is a loss of valuable information that could be exploited by the matching algorithm in order to compute the best possible placement. Indeed, a NUMA machine is most of the time hierarchically structured. A tree can thus provide a more reliable representation than a topology matrix. Since HWLOC internally uses a tree-like data structure to store all information, we can easily build another data structure derived from HWLOC's. Formally, we have a tree of hardware elements. The depth of this tree corresponds to the depth of this element in the hierarchy, cores (or other computing elements) are leaves of the tree.

3.2 Application Communication Patterns

The second vital piece of information is the target application's communication pattern. In our case, this pattern consists of the global amount of data exchanged between each pair of processes in the application and is stored in a $p \times p$ communication matrix (where p is the number of processes). This approximation yields to a "flatten" view of the communication activities that occur during an application's execution. In order to gather the needed data, we chose to introduce a slight amount of profiling elements within an existing MPI implementation (MPICH2). By modifying the low-level communication channels in the MPICH2 stack we are able to trace data exchanges in both cases of point-to-point and collective communication[5]. Since this tracing is very light, it will not disturb an application's execution. This approach has also been implemented by other vendor MPI implementations, such as HP-MPI (now Platform MPI) [3]. The main drawback is that a preliminary run of the application is mandatory in order to get the communication pattern and a change in the execution (number of processors, input data, etc.) often requires to rerun the profiling. However, there are other possible approaches. For instance, the MPIPP framework uses a tool called FAST [11] that combines static analysis of the application's code and dynamic execution of a modified application that executes much faster while retaining the same communication pattern as the original source application. The goal is to reduce the time necessary to determine the communication pattern by several magnitudes. It is to be noted that in all approaches, an application, either the original one or a simpler version, has to be executed.

[5] This profiling could also have been done by tools such as VampirTrace

3.3 Formal definition

In [2] the authors defined the problem using the communication and topology matrix formalism. Let C a $p \times p$ communication matrix (representing the amount of data exchanged between pair of processes) and T a $n \times n$, $n \geq p$, topology matrix (representing communication speed between pairs of cores), find a permutation σ of $\{1 \dots p\}$ such that $\sum_{i \neq j} C[i,j]/T[\sigma_i, \sigma_j]$ is minimized. The value σ_i tells to which core process i is mapped. In our case we will deal with a topology tree. However, we can transform this tree into a topology matrix that gives all pairwise communication speeds. We make the hypothesis that the more memory elements are shared by a pair of cores, the faster the communication will occur between them. Hence, the communication speed between two cores depends on how far they are located in the tree. We can thus derive this matrix using the distance between the corresponding leaves: the farther two cores/leaves are in the tree the slower the communication. Thanks to this topology matrix we can compute the cost of a solution using the above formula and compare our approach with the ones that only use a topology matrix.

4 The TreeMatch Algorithm

Algorithm 1: The TreeMatch Algorithm

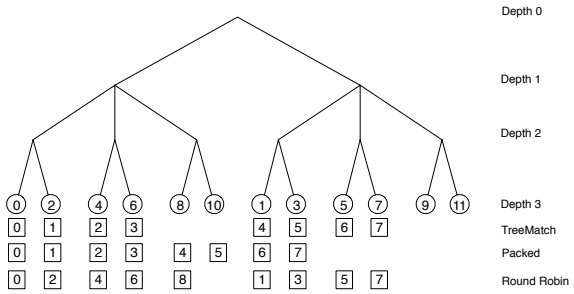
```
   Input: T// The topology tree
   Input: m // The communication matrix
   Input: D // The depth of the tree
1  groups[1..D − 1]=∅ // How nodes are grouped on each level
2  foreach depth← D − 1..1 do // We start from the leaves
3  |  p ← order of m
   |  // Extend the communication matrix if necessary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

We now describe the algorithm that we developed to compute the process placement. Our algorithm, as opposed to other approaches is able to take into account the hardware's complex hierarchy. However, in order to slightly simplify the problem, 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. The goal of the TreeMatch algorithm is to assign to each MPI process a computing element and hence a leaf of the tree. 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. The TreeMatch algorithm is depicted in Algorithm 1.

To describe how the TreeMatch algorithm works we will run it on the example given in Fig. 1. Here, the topology is modeled by a tree of depth 4 with 12 leaves (cores). The communication pattern between MPI processes is modeled by an 8×8 matrix (hence, we have 8 processes). The algorithm process the tree upward at depth 3. At this depth the arity of the node of the next level in the tree k=2, divides the order $p = 8$ of the matrix m. Hence, we directly go to line 6 where the algorithm calls function `GroupProcesses`.

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)

Fig. 1. Input Example of the TreeMatch Algorithm

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 process 2 up to 7 and so on. Formally we have $\binom{2}{8} = 5400$ 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 same 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 incompatibility, vertices are the groups and we have an edge between two vertices if the corresponding groups are incompatible. The set of groups we are looking for is hence an *independent set* of this graph. In the literature, such a graph is referred to as the complement of a Kneser Graph [8]. A valuable property[6] of the 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

[6] `http://www.princeton.edu/~jacobfox/MAT307/lecture14.pdf`

value 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 this, we valuate the graph with the amount of communication reduced thanks to this group. For instance, based on matrix m, the sum of communication of process 0 is 1114 and 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 in-approximable at a constant ratio [7]. Hence, we use heuristics to find a "good" independent set:

- **smallest values first**: we rank vertices by smallest value first and we built a maximal independent set greedily, starting by the vertices with smallest value.
- **largest value last**: we rank vertex by smallest value first and we built a maximal independent set such that the largest index of the selected vertices is minimized.
- **largest weighted degree first**: we rank vertices by their decreasing weighted degree (the average weight of their neighbours) and we built a maximal independent set greedily, starting by the vertices with largest weighted degree [7].

In our case, whatever the heuristic we use we find the independent set of minimum weight, which is {(0,1),(2,3),(4,5),(5,6)}. 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 parent.

Function AggregateComMatrix(m,g)

```
   Input: m // The communication matrix
   Input: g // list of groups of (virtual) processes to merge
1  n ← NbGroups(g)
2  for i ← 0..(n − 1) do
3      for j ← 0..(n − 1) do
4          if i = j then
5              r[i, j] ← 0
6          else
7              r[i, j] ← Σ_{i1∈g[i]} Σ_{j1∈g[j]} m[i1, j1]
8  return r
```

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)

Fig. 2. Evolution of the communication matrix at different step of the algorithm

To continue our algorithm, we will continue to build the groups at depth 2. However prior to that, we need to aggregate the matrix m with the remaining communication. The aggregated matrix is computed in the `AggregateComMatrix`. 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 Fig. 2(a)). The matrix r is of size 4 by 4 (we have 4 groups) and is returned to be affected to m (line 7 of the TreeMatch algorithm). Now, the matrix m correspond to the communication pattern between the group of processes (called virtual processes) built during this step. The goal of the remaining steps of the algorithm is to group this 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 to any other groups. This means that we add two lines and two columns full of zeroes to matrix m. The new matrix is depicted in Fig. 2(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 process 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 by 2 matrix (see Fig. 2(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, that describes a hierarchy of processes group. 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 subtree. At the end processes 0 to 7 are respectively mapped to leaves (cores) 0,2,4,6,1,3,5,7 (see bottom of Fig. 1(b)). This mapping is optimal. Indeed, it is easy to see that the algorithm provides an optimal solution if the communication matrix corresponds to a hierarchical communication pattern (processes can be arranged in tree, and the closer they are in this tree the more they communicate), that can be mapped to the topology tree (such as matrix of Fig. 1(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 subtree. On the same figure, we can see that the Round Robin mapping that maps process i on core i leads also to a very poor result.

5 Experimental Validation

In this section, we will expose several sets of results. First, we will show *simulation* performance comparisons of TreeMatch when compared to simple placement policies such as *Round Robin*, where processes are dispatched on the various NUMA nodes in a round-robin fashion, *packed*, where processes are bound onto cores in the same node until it is fully occupied and so on. We will also show comparisons between TreeMatch and the algorithm used in the MPIPP framework [2]. Basically, the MPIPP algorithm starts from a random mapping and strives to improve it by switching the cores between two processors. The algorithm stops when improvement is not possible anymore. We have implemented two versions of this randomized algorithm: MPIPP.5 when we take the best result of the MPIPP algorithm using five different initial random mappings and MPIPP.1 when only one initial mapping is used.

5.1 Experimental set-up

For performing our simulation experiments, we have used the NAS communication patterns, computed by profiling the execution of each benchmark. In this work the considered benchmarks have a size (number of processes) of 16, 32/36 or 64. The kernels are *bt*, *cg*, *ep*, *ft*, *is*, *lu*, *mg*, *sp* and the classes (size of the data) are: A for size 16 and 32/36, B, C, and D for size 32/36 and 64. When we use the benchmark, we use the topology tree constructed from the real *Bertha* machine described below.

We have also used 14 synthetic communication matrices. These are random matrices with a clear hierarchy between the processes: distinct pairs of processes communicate a lot together, then pairs of pairs communicate a little bit less, etc. For the synthetic communication matrices we also used synthetic topologies built using the Hwloc tools, in addition to the *Bertha* topology. We have used 6 different topologies with a depth from 3 to 5 mimicking current parallel machines (*e.g.* a cluster of 20 nodes with 4 socket per nodes, 6 cores per sockets, cores being paired by a L2 cache).

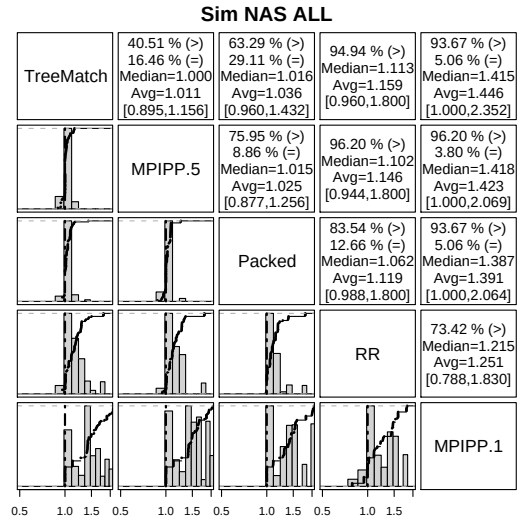
The real-scale experiments were carried out on a 96-cores machine called *Bertha*. It is a single system composed of four *NUMA nodes*. In this context, all communication between NUMA nodes is performed through shared memory. Each NUMA node features four *sockets* and each socket features six *cores*. The CPUs are Intel Dunnington at 2.66 GHz where the L2 cache is shared between two cores and the L3 is shared by all the cores on the socket. It is to be noted that the memory chipset implements some kind of L4 cache: on each NUMA node, 256 Mbytes of memory is dedicated to cache memory accesses to/from other remote NUMA nodes.

We use MPICH2 to perform our real scale experiments as its process manager (called *hydra*) includes Hwloc and thus provides a simple way to bind processes to cores.

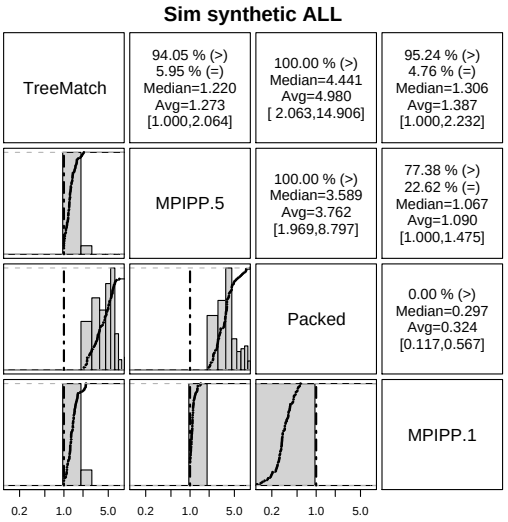
5.2 Simulation Results

We have carried-out simulation results to assess the raw performance of our algorithm.

Results are depicted in Fig. 3. On the diagonal of each figure are displayed the different heuristics. On the lower part are displayed the histogram and the ECDF (empirical cumulative distribution function) of the average simulated runtime ratio between the 2 heuristics on the corresponding row and column. If the ratio is greater than 1 the above heuristic outperforms the below heuristic. On the upper part, some numeric summary indicates: the proportion of ratios that are strictly above 1; the proportion of ratios that are equal to 1 (if any) the median ratio, the average ration and, in brackets, the maximum and minimum ratios. For example, on Fig 3(a), that TreeMatch outperforms MPIPP.1 in more than 93% of the cases with a median ration of 1.306 and an average ratio of 1.387 with a minimum ratio of 1 and a maximum ratio of 2.232.



(a) Simulation of matchings of the NAS benchmark for the different heuristics for all size



(b) Simulation of matchings of synthetic input for the different heuristics for all size. Here RR is not shown as it is identical to the Packed mapping

Fig. 3. Simulation results

On Fig. 3(a), we see that for NAS communication patterns, our TreeMatch Algorithm is better than all the other algorithms. It is only slightly better than MPIPP.5 but this version of MPIPP is run on five different seeds and hence is always slower than our algorithm[7]. The MPIPP version with one seed (MPIPP.1) is outperformed by all the other algorithms. The packed method provides better results than the Round Robin method due because cores are not numbered sequentially by the system or the BIOS.

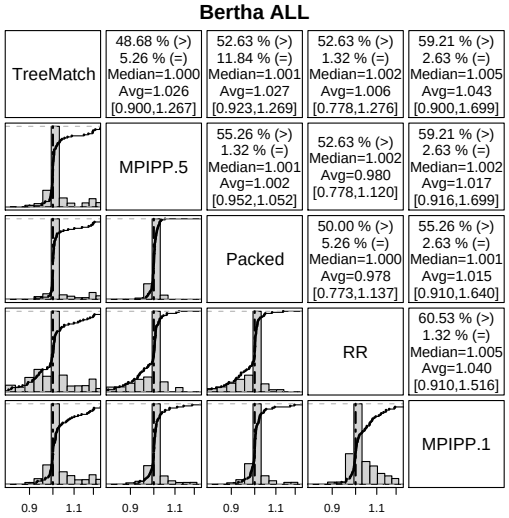
[7] up to 82 times slower in some cases

On Fig. 3(b), we see that for synthetic input, the results are even in better favor for the TreeMatch algorithm. This comes, from the fact that our algorithm finds the optimal matching for these kinds of matrices as shown in section 4.

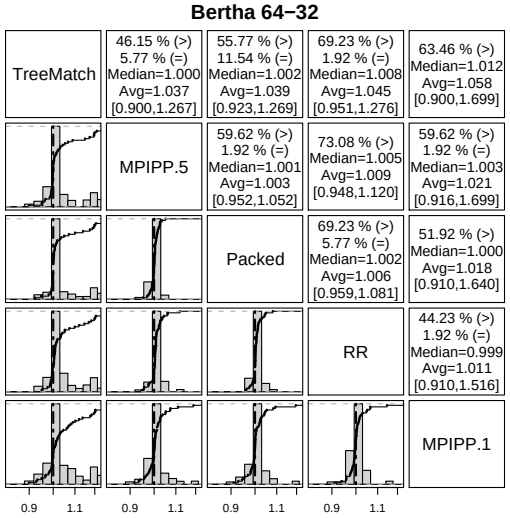
5.3 NAS Parallel Benchmarks

We then compare the results between each heuristics on the real *Bertha machine*, using the NAS benchmarks. Here, ratios are computed on average runtime of at least 4 runs.

Results are shown in Fig. 4.



(a) NAS Benchmark comparison of the different heuristics for all size



(b) NAS Benchmark comparison of the different heuristics for size 32, 36 and 64

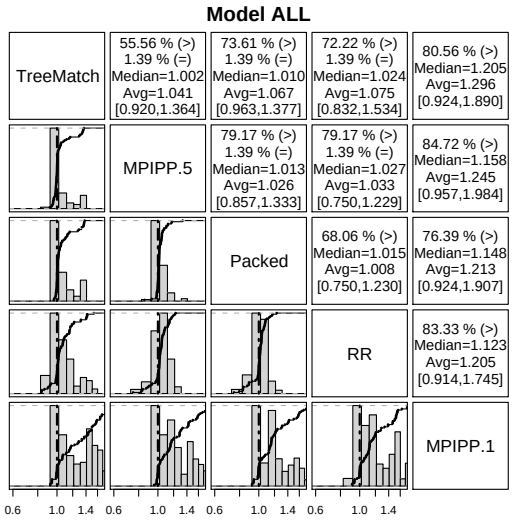
Fig. 4. NAS Comparison

In 4(a) we see that the TreeMatch is the best heuristics among the all the other tested ones. It slightly outperforms MPIPP.5, but this heuristic is much slower than ours. Surprisingly, TreeMatch is also only slightly better than Round Robin and in some cases the ratio is under 0.8. Actually, it appears than Round Robin is very good for NAS of size 16 (when all cores are grouped to the same node). This means that for small size problems a clever mapping is not required. However, if we plot the ratios for sizes above or equal to 32 (Fig. 4(b)), we see that, in such cases, TreeMatch compares even more favorably to Packed, RoundRobin or MPIPP.1 (Round-Robin, being in this case the worst method). Moreover, we see that TreeMatch is never outperformed by more than 10% (the ratio is never under 0.9) and in some cases the gain approaches 30%.

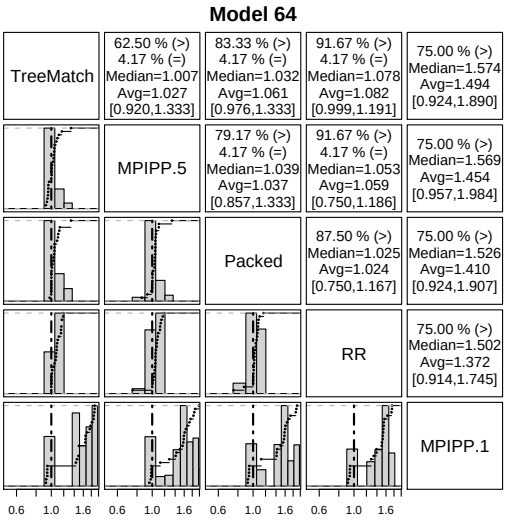
5.4 NAS Communication Patterns Modeling

In the previous section, we have seen that on the average the TreeMatch algorithm sometimes outperforms by only a small margin the other methods and we have many

comparable performances (in the histograms of the Fig. 4, most of the result have a ratio close to 1). We conjecture that this is mainly due to the fact that in some NAS kernels, the granularity is fairly high and hence the communication time is not the dominant part of the whole execution. Moreover, the communication patterns we use are an aggregation of the whole execution and do not account for phases in the algorithm where the communication affinity between processes may change. Hence, to evaluate the impact on the communication of our mapping, we have designed an MPI program that executes only the communication pattern (exchanging data corresponding to the pattern, with `MPI_Alltoallv`) and does not perform any computation. Results are displayed in Fig 5. In 5(a) we see that the TreeMatch is the best heuristic. Except for MPIPP.5, it outperforms the other heuristics in almost 75% of the cases. In several cases, the gain exceeds 30% and the loss never exceeds 10% (except for RR where the minimum ratio is 0.83). When we restrict the experiments to 64 processes the results are even more favorable to TreeMatch (Fig 5(b)). In this case, the overall worst ratio is always greater than 0.92 (8% degradation) while it outperforms the other techniques up to 20%. Moreover, it has a better or similar performance than MPIPP.5 in two thirds of the cases.



(a) NAS Benchmark communication pattern comparison of the different heuristics for all size



(b) NAS Benchmark communication pattern comparison of the different heuristics for size 64

Fig. 5. Communication pattern comparison for the NAS Benchmark

6 Conclusion and Future Works

Executing a parallel application on a modern architecture requires to carefully take into account the architectural features of the environment. Indeed, current modern parallel computers are highly hierarchical both in terms of topology (*e.g.* clusters made of nodes of several multicore processors) and in terms of data accesses or exchanges (NUMA architecture

with various levels of cache, network interconnection of nodes, etc.). In this paper, we have investigated the placement of MPI processes on these modern infrastructure. We have proposed an algorithm called TreeMatch that maps processes to computing elements based on the hierarchy topology of the target environment and on the communication pattern of the different processes. Under reasonable assumptions (e.g if the communication pattern is structured hierarchically), this algorithm provides an optimal mapping.

Simulation results show that our algorithm outperforms other approaches (such as the MPIPP algorithm) both in terms of mapping quality and computation speed. On the NAS benchmarks we have also shown that our algorithm is generally better than other approach and that the quality improves with the number of processors. As, in some cases, the TreeMatch performance is very similar to other strategy; we have studied its impact when we remove the computations. In this case, we see greater difference in terms of performance. We can then conclude that this approach delivers its full potential for applications having a huge volume of communication. However, this difference also highlights some modeling issues as the communication matrix is an aggregated view of the whole execution and does not account for different phases of the application with different communication patterns.

For instance, the same amount of data can be exchanged with a single big message of many small ones. Currently we are unable to make a distinction between such cases when it is clear that the behavior is quite different. In order to take this phenomenon into account we will have to isolate application *time slices* and remap the MPI processes during such time slices. What granularity for slices would be the most beneficial to performance? Also, we would have to modify the mapping during execution. For intranode communication this task is easy but for internode communication we would have to migrate processes from between nodes. Using virtual machines in this context might be a way to implement this.

References

1. François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond 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.
2. Hu Chen, Wenguang Chen, Jian Huang, Bob Robert, and H. Kuhn. Mpipp: an automatic profile-guided parallel process placement toolset for smp clusters and multiclusters. In Gregory K. Egan and Yoichi Muraoka, editors, *ICS*, pages 353–360. ACM, 2006.
3. David 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`.
4. Evelyn Duesterwald, Robert W. Wisniewski„Peter F. Sweeney, Gheorghe Cascaval and Stephen E. Smith. Method and System for Optimizing Communication in MPI Programs for an Execution Environment, 2008. `http://www.faqs.org/patents/app/20080288957`.
5. François Pellegrini. Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs. In *Proceedings of SHPCC'94, Knoxville*, pages 486–493. IEEE, may 1994.
6. Jesper Larsson Träff. Implementing the MPI process topology mechanism. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–14, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

7. A. Kako, T. Ono, Hirata T., and M. M. Halldorsson. Approximation algorithms for the weighted independent set problem. In *LNCS*, number 3787, pages 341–350. SPRINGER-VERLAG, 2005.
8. M. Kneser. Aufgabe 300. *Jahresber. Deutsch. Math. -Verein 58*, 1955.
9. Clyde Kruskal and Marc Snir. Cost-performance tradeoffs for communication networks. *Discrete Applied Mathematics*, 37-38:359–385, 1992.
10. Guillaume Mercier and Jérôme 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, 2009. Springer.
11. Jidong Zhai, Tianwei Sheng, Jiangzhou He, Wenguang Chen, and Weimin Zheng. Fact: fast communication trace collection for parallelapplications through program slicing. In *SC*. ACM, 2009.