

Towards an efficient process placement policy for MPI applications in multicore environments

Guillaume Mercier, Jérôme Clet-Ortega

▶ **To cite this version:**

Guillaume Mercier, Jérôme Clet-Ortega. Towards an efficient process placement policy for MPI applications in multicore environments. Europvm/mpi 2009, Sep 2009, Espoo, Finland. pp 104–115, 10.1007/978-3-642-03770-2_17 . inria-00392581v2

HAL Id: inria-00392581

https://inria.hal.science/inria-00392581v2

Submitted on 26 Nov 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Towards an Efficient Process Placement Policy for MPI Applications in Multicore Environments

Guillaume Mercier and Jérôme Clet-Ortega

Université de Bordeaux - INRIA - LaBRI
351, cours de la Libération F-33405 Talence cedex.
{guillaume.mercier, jerome.clet-ortega}@labri.fr

Abstract. This paper presents a method to efficiently place MPI processes on multicore machines. Since MPI implementations often feature efficient supports for both shared-memory and network communication, an adequate placement policy is a crucial step to improve applications performance. As a case study, we show the results obtained for several NAS computing kernels and explain how the policy influences overall performance. In particular, we found out that a policy merely increasing the intranode communication ratio is not enough and that cache utilization is also an influential factor. A more sophisticated policy (eg. one taking into account the architecture's memory structure) is required to observe performance improvements.

Keywords : *Message-Passing*, *Multicore architectures*, *Process placement.*

1 Introduction

In the last decade, parallel computer architectures have evolved dramatically. Clusters have shifted from an assembly of uniprocessor machines interconnected by a single network to a complex and highly hierarchical structure. Nodes are now composed of several multicore processors sharing memory banks physically scattered across the node. Major CPU manufacturers like AMD (HyperTransport) and Intel (Quick Path Interconnect) follow this trend. The memory access time depends on the location of both the core and the memory bank. This is often designed as the Non-Uniform Memory Access (NUMA) effect. The memory hierarchy is also more complex due to the increase of cache levels. This sharing of memory resources depends on the CPU architecture and differs from a manufacturer to another. As for the interconnection network, multirail systems where several high-speed NICS are connected to a node are sometimes also encountered.

A real challenge for a parallel applications is to exploit such architectures at their full potential. In order to achieve the best performance, many factors must be taken into consideration and studied. The first one is to make use of an implementation of the MPI specifications [1] able to efficiently take advantage of a multicore environment. Whilst the MPI standard is architecture-independent, it is an implementation's task to bridge the gap between the hardware's performance and the application's. Indeed, recent MPI-2 implementations such as Open MPI [2] or MPICH2 [3] fulfill this purpose and offer a very satisfactory performance level on multicore architectures.

However, in order for an MPI implementation to fully exploit the underlying hardware, the MPI application processes have to be placed carefully on the cores of the target architecture. This placement policy has to be defined by both the application's communication pattern and the hardware's characteristics. For instance, if some application processes communicate more frequently than others, they should be regrouped and placed on the same multicore node. By doing so, the amount of intranode communication increases and the application global performance will improve since intranode communication (shared memory) is faster than internode communication (network).

In this paper we expose the method and software tools we employed to allow an MPI application to better take advantage of a multicore environment. We will show a performance improvement not due to modifications of the MPI implementation itself but rather due to a relevent process placement. The rest of this paper is organized as follows: Section 2 describes how process placement is determined and on which set of tools and algorithms it relies. Experimentals results are presented in Section 3 with performance figures for some of the NAS computing kernels. Section 4 lists related works in this area of MPI process placement and discusses some issues raised by information gathering. Section 5 concludes this paper and opens future perspectives.

2 Computation of a relevant MPI process placement

In order to place the MPI processes in a relevant fashion, we have to gather information about the target architecture and the application's communication pattern. Once both are available we analyze them and determine the best possible placement. The criterion choice should follow a user-defined strategy. However, the current scope of this work does not encompass all MPI applications.

2.1 Hypotheses about MPI applications and their execution environment

In the rest of this paper we consider *static* MPI applications. By static, we mean that the application does not use any of the dynamic processes features offered by MPI-2. We also exclude hybrid MPI applications that rely on multithreading features (such as OpenMP directives). The number of computing entities (threads and processes) is therefore guaranteed to remain constant during an application's execution. We also consider the target machine to be fully dedicated to the MPI application. All cores are usable by the MPI processes with the restriction that only a single MPI process runs on a given core. As a consequence of these points, a static mapping between the MPI processes and the CPU cores can be computed before launching the application. This placement will not need to be modified during the application's execution.

2.2 Gathering the hardware's information

As previously explained, clusters of NUMA nodes are hierarchically structured. For instance, figure 1 shows the architecture of an AMD Opteron-based compute

node. This compute node is composed of four dies with two cores each. Each die possesses a set of main memory banks attached to it. A core features its own Level 1 cache (not shown) and Level 2 cache, not shared with the other core on the same die. A core located on a die can access the main memory of any other die but the access time increases as the physical distance between the core and the memory bank lengthens. A network interface card (NIC) can be attached to a bus connected to Die #0 and Die #1 . Since the cores located on these dies are physically closer to the I/O bus, one might expect faster network transactions for processes mapped on these cores.

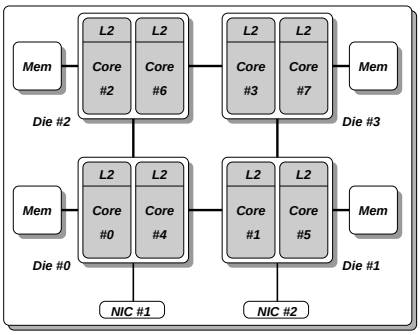


Fig. 1. An exemple of hierarchical compute node: an 8-cores Opteron

Some tools can provide us with the needed information (e.g libtopology or Portable Linux Processor Affinity [4] available for Linux). However, such tools are not portable and accurate enough over a wide spectrum of operating systems. To gather the hardware's information, we used a topology discovery mechanism implemented in the PM^2 runtime system [5]. This feature is employed to help the scheduling of threads on multicore nodes [6]. Currently, this mechanism only deals with CPUs and does not deliver information about networks and I/O buses. For instance, when applied to the Opteron node as depicted by figure 1, the PM^2 topology discovery mechanism outputs the information displayed by figure 2.

```
Machine:
 NUMANode + Die: Node#0(8GB) Die#0
  L2Cache + Core + L1Cache + SMTproc : L2#0(1MB) Core#0 L1#0(64kB) CPU#0
  L2Cache + Core + L1Cache + SMTproc : L2#4(1MB) Core#1 L1#4(64kB) CPU#4
 NUMANode + Die: Node#1(8GB) Die#1
  L2Cache + Core + L1Cache + SMTproc : L2#1(1MB) Core#0 L1#1(64kB) CPU#1
  L2Cache + Core + L1Cache + SMTproc : L2#5(1MB) Core#1 L1#5(64kB) CPU#5
 NUMANode + Die: Node#2(8GB) Die#2
  L2Cache + Core + L1Cache + SMTproc : L2#2(1MB) Core#0 L1#2(64kB) CPU#2
  L2Cache + Core + L1Cache + SMTproc : L2#6(1MB) Core#1 L1#6(64kB) CPU#6
 NUMANode + Die: Node#3(8GB) Die#3
  L2Cache + Core + L1Cache + SMTproc : L2#3(1MB) Core#0 L1#3(64kB) CPU#3
  L2Cache + Core + L1Cache + SMTproc : L2#7(1MB) Core#1 L1#7(64kB) CPU#7
```

Fig. 2. Hardware information generated by PM^2's topology discovery mechanism.

We generate from this ouptut a data structure that other tools will use to compute the placement. This data structure is a graph with vertices representing the CPU cores and weighted edges. This graph is complete and non-oriented. The weight affected to each edge increases as more elements of the memory hierarchy are shared between cores. It is also affected by the NUMA effect.

Figure 3 shows the various weight values chosen for the Opteron compute node (figure 1). We decided to put the largest weight for cores on the same die because they directly share memory banks. The next largest weight value corresponds to NUMA effects: for instance core #0 is able to access the memory attached to Die #2 and Die #1 faster than the memory attached to Die #3. When the machine is made of several compute nodes, we build a larger graph and affect a larger weight for cores located within the same compute node. The smallest weight values thus correspond to communication using the network.

$$Machine = \begin{pmatrix} 0 & 100 & 100 & 10 & 1000 & 100 & 100 & 10 \\ 100 & 0 & 10 & 100 & 100 & 1000 & 10 & 100 \\ 100 & 10 & 0 & 100 & 100 & 10 & 1000 & 100 \\ 10 & 100 & 100 & 0 & 10 & 100 & 100 & 1000 \\ 1000 & 100 & 100 & 10 & 0 & 100 & 100 & 10 \\ 100 & 1000 & 10 & 100 & 100 & 0 & 10 & 100 \\ 100 & 10 & 1000 & 100 & 100 & 10 & 0 & 100 \\ 10 & 100 & 100 & 1000 & 10 & 100 & 100 & 0 \end{pmatrix}$$

Fig. 3. Matrix representation of the hardware's information from figure 2. (*Machine(i,j)* represents the potential of communication between cores #i and #j).

2.3 Collecting the application's communication pattern data

The second piece of information deals with the application's communication pattern. Each application possesses its own pattern influenced by the number of participating processes. Our chosen characterization criterion for this pattern is the amount of MPI data exchanged between processes. Therefore, we need to compute this amount for each pair of processes in the application.

Several sophisticated tools are provided for MPI application tracing and analysis, such as the MPI Parallel Environment (MPE). However, they do not provide all the necessary information. For instance, MPE is able to trace all calls to MPI routines made by the application. By analyzing such a trace and focusing on the point-to-point calls, we can have hints about the communication pattern. This approach is simple and require to configure the MPI implementation with MPE support and to link the application with the appropriate libraries. This is limited by two factors: first, the trace generated can be potentially very large and second, the amount of data exchanged in collective operations is not taken into account.

As a consequence we found simpler and more accurate to modify an MPI implementation to collect the desired information. By modifying directly an implementation, we reduce drastically the trace size and take into account collective communication operations. In order to get a generic (that is, not implementation-specific) information, we trace only the size of MPI user data exchanged between

processes. All costs induced by the implementation's internal protocols are not counted. As for the hardware's data, we represent this communication pattern with a complete, non-oriented graph with weighted edges. In this case, the weight value increases as the amount of data exchanged between MPI processes grows. An example is depicted by figure 4: the matrix represents the communication pattern for one NAS benchmark (lu.B.8).

$$Application = \begin{pmatrix} 0 & 1000 & 10 & 1 & 100 & 1 & 1 & 1 \\ 1000 & 0 & 1000 & 1 & 1 & 100 & 1 & 1 \\ 10 & 1000 & 0 & 1000 & 1 & 1 & 100 & 1 \\ 1 & 1 & 1000 & 0 & 1 & 1 & 1 & 100 \\ 100 & 1 & 1 & 1 & 0 & 1000 & 10 & 1 \\ 1 & 100 & 1 & 1 & 1000 & 0 & 1000 & 1 \\ 1 & 1 & 100 & 1 & 10 & 1000 & 0 & 1000 \\ 1 & 1 & 1 & 100 & 1 & 1 & 1000 & 0 \end{pmatrix}$$

Fig. 4. Matrix representation of NAS LU (class B, 8 processes) communication pattern. (*Application(i,j)* represents the amount of communication between processes #i and #j).

2.4 Mapping a MPI process rank to a CPU core number

The final step is to extract an embedding of the application's graph from the target machine's graph. We use the *Scotch* software [7] to solve this *NP* graph problem. Scotch applies graph theory, with a divide and conquer approach, to scientific computing problems such as graph and mesh partitioning, static mapping, and sparse matrix ordering. In our case, we use the ability to construct a static mapping. Scotch implements dual recursive bipartitioning algorithms to perform this task [8] and computes static mappings for graphs larger than 2^{32} vertices. This ensures that we can create a mapping for all MPI applications, regardless of their size.

`MPI_COMM_WORLD` Rank	0	1	2	3	4	5	6	7
Core Number	3	7	4	0	6	2	5	1

Table 1. Resulting mapping for application NAS lu.B.8 on the Opteron compute node.

For instance, table 1 gives the static mapping computed by Scotch in the case of the NAS lu.B.8 benchmark (figure 4) launched on the Opteron compute node (figure 3). Using this static mapping, we finally generate a specific command line fully customized for each *(Application, Target Machine)* couple. Practically, each MPI process is affected to its dedicated core with the `numactl` command.

3 Evaluation: a case study with NAS computing kernels

In this section, we present some results obtained by applying the method previously described on several NAS computing kernels. We first describe our experimental environment, then show the results and finally comment on them.

3.1 Experimental environment and NAS benchmarks choice

We carried out experiments on a 10-nodes cluster called *Borderline*, part of the Grid5000 testbed [9]. Borderline nodes are similar to the exemple depicted by figure 1: each node features four dies (a 2.6 GHz AMD Opteron 2218) with two cores each. A core possesses its own Level 2 cache (1 MBytes) not shared with the other core on the same die. The total amount of memory is 32 GBytes per node (8 GBytes per die). A Myrinet 10G NIC is attached on a bus to Die #0. The Linux kernel installed is 2.6.22.1 (SMP version).

We benchmark some of the NAS computing kernels in order to assess the relevance of our placement method and policy. We first make a run of all NAS benchmarks to gather their respective communication pattern data, as described in section 2.3. Since our placement technique is currently based only on the global amount of data exchanged, we only run tests where this amount is significant: 64-processes jobs of classes C and D. Likewise, we test only applications with irregular communication patterns. Indeed, they are likely to be the most influenced by the placement of processes in a multicore environment. Among all NAS kernels, only BT, CG, LU, MG and SP have an irregular communication pattern. In the case of CG, some processes do not communicate with others. But when communication occurs, the global amount of data exchanged for each pair of processes is of same magnitude. In the case of BT, LU, MG and SP these amounts differ dramatically from one pair to the other.

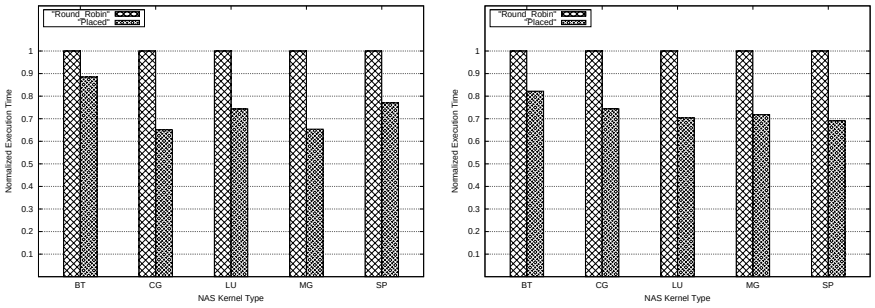
3.2 NAS performance results for two MPI implementations

In order to confirm that the method used and the data gathered are implementation-independent, we expose performance comparisons for two different MPI implementations. This first one is MPICH2-Nemesis, which relies on a very efficient intranode communication system using shared-memory [10]. We configured Nemesis to use its most recent MX support (available in the MPICH2 1.1 release). The other MPI implementation is MPICH2-MX [11] designed and implemented by Myricom. This implementation also features an efficient intranode communication system based on an in-kernel mechanism.

	BT		**CG**		**LU**		**MG**		**SP**	
	C	D	C	D	C	D	C	D	C	D
Round-Robin	51.6	1038.6	23.9	1177.5	45.5	1356.5	5.6	119.2	78.6	2015.63
Placed	45.6	851.7	15.6	848.4	33.6	938.3	3.7	85.3	60.2	1386.8

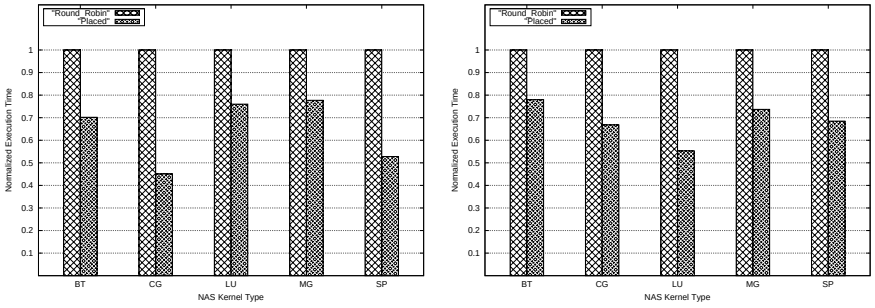
Table 2. MPICH2-Nemesis:MX execution times in seconds for two different placement policies.

Table 2, figures 5 and 6 show the NAS performance for two different placement policies. The first placement policy is a simple *Round-Robin*-type policy where the process number i is executed on the node number n where $n \equiv i \mod 8$ (in our case each node features eight cores). This type of placement is



(a) Class C - 64 Processes (b) Class D - 64 Processes

Fig. 5. Process placements comparison for MPICH2-Nemesis:MX.



(a) Class C - 64 Processes (b) Class D - 64 Processes

Fig. 6. Process placements comparison for MPICH2-MX.

commonly used when launching MPI applications without any knowledge of their communication patterns. With this scheme, the operating system chooses on which core a process is executed. The other policy, called *Placed* is the one determined thanks to the method described in section 2. In the case of MPICH2-Nemesis:MX, there is a significant performance gap between the *Placed* policy and the *Round-Robin* policy. This result applies to both C and D classes. As our results indicate, this improvement for Classes C and D kernels is roughly of 25%. In the case of Myricom's MPICH2-MX, this gap is even larger: execution times are 34% faster when our *Placed* policy is enforced. One might note that in the case of the CG kernel (class C), performance is much better with our custom placement than with the *Round-Robin* one.

3.3 NAS performance and placement policies analysis

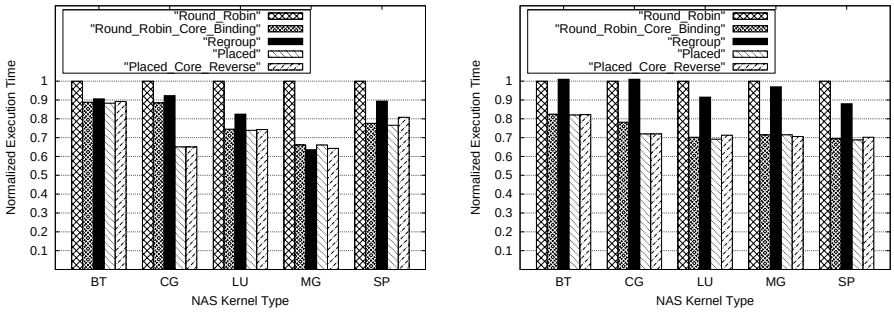
The *Placed* policy regroups MPI processes on the same node as much as possible. The ratio of intranode communication (using shared-memory) versus internode

communication (using the Myrinet network) increases as Table 3 shows. However, questions remain: is this performance gap solely due to this increase of intranode communication ratio in the application? Do others factors – such as memory hierarchy and structure – influence performance too? In order to answer

	BT		**CG**		**LU**		**MG**		**SP**	
	C	D	C	D	C	D	C	D	C	D
Round-Robin	32%	33%	0%	0%	46%	46%	39%	50%	31%	33%
Placed	53%	54%	78%	78%	70%	70%	55%	66%	52%	54%

Table 3. Overall intranode communication ratios in NAS kernels for two different placement policies.

these questions, we run the NAS kernels with three other placement policies. The first new policy is the same as the *Round Robin* policy, except that each process is bound to a particular core with the `numactl` command. In this way, the operating system cannot change a process location when scheduling occurs. The Level 2 cache utilization is therefore better. In this case, the intranode communication ratio is not improved. The second new policy regroups the MPI processes on the nodes as much as possible (like in the *Placed* policy) but instead of binding each MPI process on its dedicated core, we let the operating system choose the placement (like in the regular *Round-Robin* policy). By doing so, the overall ratio of intranode communication versus internode communication is increased but we do not take anymore into account factors such as NUMA effects, memory hierarchy or die sharing by processes. Cache utilization will be negatively impacted when processes are placed according to this policy. The last policy also regroups process on the compute nodes but we force the intranode placement to be suboptimal: in this case we place the processes that communicate the most on opposite dies in the compute node. This last policy and the *Regroup* policy share a common point in that process placement within the node is very poor. The difference between the two is that with the *Placed Core Reverse* policy, cache utilization is better since processes are pinned to a core. Figure 7 shows the results: both *Round-Robin* and *Placed* policies are the same as in section 3.2, while *Round-Robin Core Binding*, *Regroup* and *Placed Core Reverse* represent the new placement policies described above. The results indicate that merely improving the intranode communication ratio is not enough to deliver better performance. Increasing this ratio without taking into account cache utilization leads to suboptimal results. Globally, our *Placed* policy delivers slightly better results than the others policies. The CG kernel is the benchmark for which the performance improvement is the most noticeable. These results suggest that cache utilization has a great influence on NAS performance. and advocates for a placement policy that takes architectural factors into consideration. However, we think that the Opteron compute nodes we used have a too small NUMA effect for it to impact applications performance. Also, the results obtained with the *Placed Core Reverse* policy show that despite the increase of traffic on the memory bus (induced by this policy), the impact on performance is limited on



(a) Class C - 64 Processes

(b) Class D - 64 Processes

Fig. 7. Process placements comparison for MPICH2-Nemesis:MX.

the compute nodes we used. Last, Level 2 cache is not shared between cores on the same die as explained in section 2.2. Those three aspects lead us to think that our placement policy could perform even better on nodes featuring more cores, with substantial NUMA effects, and where cache is effectively shared between cores. Finally, since we do not know the theoretical achievable performance of this particular cluster, we cannot assess how close to the optimal placement we actually are.

4 Related Works and Discussion

An adequate process placement is a necessity to fully take advantage of multicore environments. The MPI standard itself offers a set of routines that allow the applications developers to create and manipulate *topologies*. Using such topologies, MPI processes could be placed according application's communication pattern. However, only a subset of existing MPI applications makes use of these topology creation and management features. Also, as [12] points out, not all MPI implementations support these efficiently. At last, this mapping between the *virtual* topology created and the *physical* topology (the issue on which we focus in this paper) is outside the scope of the MPI standard. Our approach – deploying the MPI processes according to a matching between the application's communication pattern and the machine's architecture – is more generic and can benefit to any MPI application. Actually, this approach is not MPI-dependent, but rather *message-passing*-dependent. It can be applied even to non-MPI applications as long as the necessary information is collectable.

Placing MPI processes according to the underlying target architecture using graph theory has already been explored. Several vendor MPI implementations, such as the ones provided by Hewlett-Packard [13] or by IBM (according to [14]) make use of such mechanism. [12] also formalizes the problem with graphs. In these cases, however, the algorithm computing the final mapping is MPI implementation-specific. In our framework, we rely on an external piece of software (Scotch) fully tailored for graph computations. This ensures both performance and scalability. Scotch is indeed able to work on very large classes of

graphs, larger that the maximum number of MPI processes present in any MPI application.

As explained in section 2.2, architecture's information is gathered thanks to PM^2's topology discovery mechanism. But this feature is fully embedded in this software stack and rather unconvenient to use. Such a topology discovery feature would be very useful for many other applications, especially MPI process managers. Hybrid MPI+OpenMP applications could also take advantage from it[15]. There is a clear need for a portable and accurate tool dedicated to topology discovery. We already started to work on this specific point.

The last issue we would like to address regards the gathering of applications communication patterns. In order to get the needed information to create the mapping, we have to execute a prior run of the applications compiled with a modified MPI implementation fitting our needs. Other works in this field use the same coarse scheme. They also emphasize that tracing tools could be enhanced to address the specific issue of getting the amount of data exchanged between two given processes, as HP's *Light Weight Instrumentation* [13] does. The necessary information could be an *estimate* of the communication flows that would allow us to *rank* pairs of processes accordingly. However, the ability to perform this prior run does not systematically exists. Other solutions should be investigated. For instance, relying on an application's programmer's knowledge is also possible but once again far from always possible. Could this kind of information be computed at *compile* time? Would it be possible to pass an option to the `mpicc` compiler that would *automatically* generate the customized command line? These are issues we would like to address in the near future.

5 Conclusion and Future Work

In this paper, we expose a method that leads an MPI application to better exploit its target architecture, especially complex and hierarchical multicore environments. This method relies on gathering information about the underlying hardware and the application's communication pattern. This information is then used to create a mapping between MPI process ranks and each node's core numbers. Finally an application-specific command line is generated. Our method uses free, open-source software and is not tightly integrated within a particular MPI implementation. Being able to place the processes according to the marchine's topology increases performance. We analyzed several placement policies and found out that merely increasing the intranode communication ratio in an application is not enough to deliver more performance: an adequate cache utilization is also mandatory to enhance performance. The current results are mitigated: our sophisticated placement policy improves MPI performance applications but should yield even better results on more complex architectures. We look forward to run our experiments on a different (and more complex) class of hardware.

Possible future works are numerous: as we previously stated, we began to work on a software tool that would allow us to easily extract hardware's information. Integrating information about I/O buses or even GPUs should also be considered. We would like to relax some constraints about the type of MPI

applications falling into the scope of this work. We do consider *static* MPI applications, that is, applications where the amount of computing entities remains constant throughout the execution. This excludes both applications spawning new MPI processes and multithreaded ones. Since hybrid programming, mixing message-passing and multithreading, is considered as a possible way to better program multicore architectures, we plan to address the issue of MPI processes placement when OpenMP parallel sections appear in MPI processes. Another interesting direction would be to refine the application data regarding its communication pattern. For now, we did only consider a *spatial* pattern. But what about the *temporal* pattern? Indeed, the data exchanges occuring between a given pair of processes may vary during the application's execution. 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 one node to the other. Using virtual machines in this context might be a way to implement this. Also, some other influential factors such as contention on the memory and I/O buses could be taken more into consideration. This would lead to an even more refined placement policy but requires to comprehend thoroughly the target application. Generating MPI-implementation dependent information could also be considered in order to increase the policy's accuracy. Last, we plan to investigate the feasability of gathering the application's information at compile time.

Acknowledgments
Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).
The authors wish to thank Sébastien Fourestier, our Scotch Guru.

References

1. Message Passing Interface Forum: MPI-2: Extensions to the message-passing interface. http://www.mpi-forum.org/docs/mpi-20.ps (1997)
2. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: Goals, concept, and design of a next generation MPI implementation. In: Proceedings of the 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary (2004) 97–104
3. Argonne National Laboratory: MPICH2. http://www.mcs.anl.gov/mpi/(2004)
4. Jeff Squyres and al.: Portable Linux Processor Affinity (2008) `http://www.open-mpi.org/projects/plpa/`.
5. Raymond Namyst, Yves Denneulin, Jean-Marc Geib and Jean-François Méhaut: Utilisation des processus légers pour le calcul parallèle distribué : l'approche PM2. Calculateurs Parallèles, Réseaux et Systèmes répartis **10** (1998) 237–258
6. Samuel Thibault, Raymond Namyst and Pierre-André Wacrenier: Building Portable Thread Schedulers for Hierarchical Multiprocessors: the BubbleSched Framework. In: EuroPar, Rennes,France, ACM (2007)

7. François Pellegrini: Scotch and LibScotch 5.1 User's Guide. ScAlApplix project, INRIA Bordeaux – Sud-Ouest, ENSEIRB & LaBRI, UMR CNRS 5800. (2008) http://www.labri.fr/perso/pelegrin/scotch/.
8. François Pellegrini: Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs. In: Proceedings of SHPCC'94, Knoxville, IEEE (1994) 486–493
9. R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lantri, J. Leduc, N. Melab, G. Mornet,R. Namyst, P. Primet, B. Quetier, O. Richard, E-G. Talbi and I. Touche: Grid'5000: a large scale and highly reconfigurable experimental Grid testbed. International Journal of High Performance Computing Applications **20** (2006) 481–494
10. Darius Buntinas, Guillaume Mercier and William 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** (2007) 634–644
11. Myricom: MPICH2-MX (2009) http://www.myri.com/scs/download-mpichmx.html.
12. Jesper Larsson Träff: Implementing the MPI process topology mechanism. In: Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing, Los Alamitos, CA, USA, IEEE Computer Society Press (2002) 1–14
13. 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.
14. 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.
15. Rolf Rabenseifner, Georg Hager and Gabriele Jost: Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. In: Proceedings of the 17th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP 2009), Weimar, Germany (2009) 427–436