Improving MPI Applications Performance on Multicore Clusters with Rank Reordering

Guillaume Mercier and Emmanuel Jeannot

Université de Bordeaux, INRIA, LaBRI
351, cours de la Libération F-33405 Talence, France
{guillaume.mercier,emmanuel.jeannot}@labri.fr

Abstract. Modern hardware architectures featuring multicores and a complex memory hierarchy raise challenges that need to be addressed by parallel applications programmers. It is therefore tempting to adapt an application communication pattern to the characteristics of the underlying hardware. The MPI standard features several functions that allow the ranks of MPI processes to be reordered according to a graph attached to a newly created communicator. In this paper, we explain how the MPICH2 implementation of the `MPI_Dist_graph_create` function was modified to reorder the MPI process ranks to create a match between the application communication pattern and the hardware topology. The experimental results on a multicore cluster show that improvements can be achieved as long as the application communication pattern is expressed by a relevant metric.

Keywords: Message-Passing, multicore architectures, process placement, rank reordering, communication pattern.

1 Introduction

Parallel programming is the prevalent paradigm for scientific applications. It is widely considered as the sole mean to achieve the computing power sought after by applications. Programming standards and their implementations play here a pivotal role because their efficiency conditions the overall performance. Among the parallel programming standards, the *Message Programming Interface* (MPI) is very popular because of its rich interface. Also, the implementations available manage to bridge the performance gap between the hardware and the applications. As for the hardware, the most widespread architecture to build parallel computers is based on the cluster paradigm. This trend has gained a huge momentum since its inception more than a decade ago and is still very strong. The machines used to build clusters have however changed from SMP-based nodes to more complex multicore ones, altering the way applications should be programmed. To harness such architectures is a difficult undertaking. NUMA effects, memory hierarchies and cores/cpus physical location within a node force the programmer to finely apprehend the hardware. The side effect is a decrease of performance portability: whilst any MPI code will run on such machines, only

Y. Cotronis et al. (Eds.): EuroMPI 2011, LNCS 6960, pp. 39–49, 2011.

those specifically tailored to fit the hardware will benefit from its full performance. MPI being hardware-agnostic, no function in the interface can help the programmer to retrieve information about the hardware and convey it up to the application. Some workarounds do exist at best: process managers can enforce the binding of MPI processes onto specific cores and the logical topology mechanism can be used to communicate application-specific information (such as a communication pattern for instance) to the implementation. Concerning the latter point, implementations that go beyond a trivial work are not aplenty. Only some vendors MPI implementations (such as the ones provided by HP [1] or NEC [2]) are tailored for specific classes of hardware and propose topology routines implementations taking advantage of the underlying specific fabric. Generic MPI implementations addressing a wider spectrum of hardware however manage to feature optimizations taking advantage of multicore nodes. Indeed, collective communication operations are usually designed and implemented in a hierarchical, two-levels, fashion so as to yield better performance [3]. But nothing is done in the topology mechanisms department to allow the programmer to map an application communication pattern onto the underlying hardware. In this paper, we propose an enhanced implementation of one MPI function: `MPI_Dist_graph_create`. In our expanded version, the ranks of the MPI processes calling this function are reordered to allow an application communication pattern to match as best as possible the underlying physical topology of a multicore cluster. This paper is organized as follows: Section 2 will expose the issue of mapping a communication pattern onto a hardware architecture and compares different existing techniques. Technical details are discussed in Section 3 while results are analyzed in Section 4. Section 5 will describe previous existing works while Section 6 concludes this paper and discusses future directions.

2 Matching a Communication Pattern to the Hardware Architecture: Issues and Techniques

2.1 General Overview of the Problem

During an MPI application, data are exchanged among the various participating *processes*. The MPI programming paradigm is flat: each process may communicate with any other in the application. However, depending on pairs of processes, the amount of data sent and received (in either terms of bytes/volume or number of messages) may be irregular. Hence, each MPI application possesses a so-called *communication pattern* which can be considered as an intrisic characteristic [4] of the *affinity* between processes (here, we assume that this pattern is deterministic and does not change between executions). On the other hand, the communication channels in a multicore, NUMA nodes-based cluster are heterogeneous. Internode communication using a network is slower than intranode communication using shared memory. The novelty with multicore NUMA nodes is that communication performance is also heterogeneous within the node itself. The various levels of cache memory and the NUMA effects when accessing the main

memory induce this. It is therefore rather *intuitive* to seek to adapt a potentially irregular communication pattern to the also heterogeneous (performance-wise) underlying hardware architecture.

2.2 Core Binding vs. Rank Reordering

There are two different methods to achieve this goal. The first one is called the *core binding* technique [5]. A binding algorithm determines on which physical core a specific MPI process should be located and pinned, so as to improve the overall communication performance (*e.g.* MPI processes are mapped according to the communication pattern and the hardware topology so as to minimize communication cost). An MPI application does not need to be modified: this binding information is provided by the user to the process manager which in turn enforces this user-defined binding policy at runtime. Legacy MPI applications can thus take advantage of this technique, if sufficient information is provided to the binding algorithm (which might imply an instrumentation of the application code, for example to build the communication pattern). However, this approach lacks transparency since the user has to use MPI implementation-specific command line options. Also, modifying, in a standard fashion, the binding during the course of an application is difficult. With the second method, called *rank reordering*, a new communicator is created with application-specific information attached to it. Ranks of the MPI processes belonging to this communicator can be *reordered*, meaning that they can be changed to fit some application constraints. Thus, the ranks of the MPI processes belonging to this newly created communicator could be determined to match the communication pattern to the underlying physical architecture. A reordering algorithm is necessary, playing a similar role as the binding algorithm of the first method. Legacy MPI applications would have to be slightly modified to issue a call to the ranks reordering MPI routine and use the newly produced communicator. Such reordering should be performed before application data is loaded into the MPI processes, otherwise data movements would be necessary. However, relying on a standard MPI call ensures portability, transparency and dynamicity since it can be issued multiple times during an application execution. These aspects aside, both methods yield the same performance improvements.

3 A Non-trivial Implementation of `MPI_Dist_graph_create`

This paper focuses on the *rank reordering* technique. Several MPI functions can reorder processes ranks. It is the case of `MPI_Dist_graph_create`, part of the standard since MPI 2.2 [6]. This function is meant to replace the non-scalable `MPI_Graph_map` function. `MPI_Dist_graph_create` takes as arguments a set of pointers (`sources`, `destinations`, `degrees` and `weights`) that characterize a graph. Hence, random application communication patterns can be passed to the implementation using these pointers. We modified the current

`MPI_Dist_graph_create` implementation available in MPICH2 [7] in order to allow the given input graph to be mapped onto another graph we build and that describes the underlying architecture. Such a problem is known as a *graph embedding problem*. In our case, the optimization criterion is the minimization of communication costs. Our approach is three-fold: first we gather information about the hardware topology, then we access the application communication pattern and at last we solve our graph embedding problem with a tailored algorithm.

3.1 Gathering the Hardware Information

To gather hardware information raises portability issues because we need to address the largest possible spectrum of architectures. No standard tool currently exist to perform this task. Our version of `MPI_Dist_graph_create` uses to the Hwloc library (version 1.1.1) [8] that offers a generic and portable interface to retrieve hardware information. Thanks to Hwloc, we manage to gain insights of a NUMA node structure (e.g cache hierarchies, number of processors, location of processing units within sockets, etc.). On each multicore node, one process extracts the hardware information, then a global root process gathers all these data. That is, our current implementation is *centralized*, which might impact scalability. Hwloc being currently unable to provide us information about the network topology, we consider it as flat, as in the MPI model. Now, this information has to be represented in a convenient way. Since multicore nodes are organized hierarchically, a relevant data structure is a *tree*, where leaves represent processing units. To create the data structure that represents a cluster of multicore nodes is a straightforward process: we add a new level encompassing all the subtrees representing the various NUMA nodes at the top level of the structure. This corresponds to our vision of a flat network topology.

3.2 Communication Pattern Information and Metrics

There are two cases to consider for an application communication pattern, First, newly developed MPI applications can directly use `MPI_Dist_graph_create`. In this case, the programmer has to provide the pattern information thanks to the function arguments. Indeed, the programmer is supposed to possess some knowledge of the organization of communication. But it is not always the case, especially when using collective communication, because the pattern will depend on algorithms known only by the designers of the MPI implementation. Hence, switching from one MPI implementation to another is likely to influence the application pattern. In the case of applications for which the pattern is unknown to the user, some information can be gathered by the means of instrumentation. Therefore, we introduced a lightweigt trace system in MPICH2 to retrieve the pattern information. We trace the data exchanged at the MPI application level to obtain the most implementation-independent data. Of course, a prior execution of the application is mandatory to generate a pattern data file. It contains

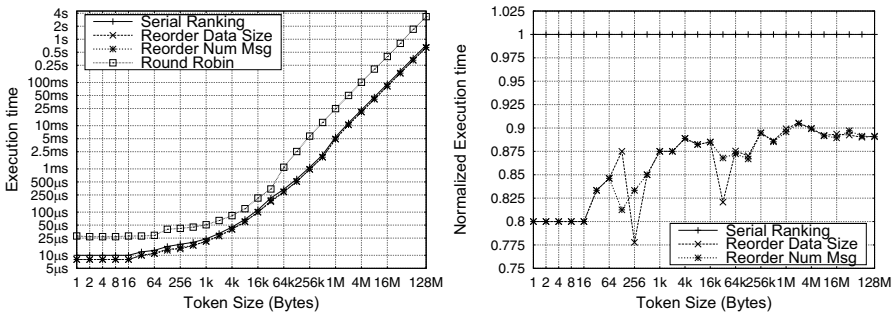
information for each pair of processes.We actually use two different *metrics* to assign weights to the edges of the pattern graph. The first metric is the global amount of data (a.k.a *Data Size*) while the second one is the number of exchanged messages (a.k.a *Number of Messages*). We also implemented a helper routine that directly reads the pattern file output by our trace system in order to fill the arguments of `MPI_Dist_graph_create` according to the chosen metric.

3.3 The TreeMatch Matching Algorithm

In order to solve our *graph embedding problem*, we implemented a new algorithm called TreeMatch [9]. The TreeMatch algorithm is a graph algorithm which takes into account the affinity of the processes expressed as a communication matrix to bind these processes to the topology. It works recursively on each level of the memory hierarchy (following a bottom-up approach) and groups processes in such a way that the cost of remaining communications is minimized. TreeMatch extracts a tree from the communication matrix representing a communication pattern and matches this tree to the hardware topology tree. Finally, the algorithm outputs a permutation of the processes σ such that process rank i (in the original communicator) is mapped on core σ_i. This algorithm is called by the global root process that possesses both the hardware information and the pattern information: indeed, the implementation is currently fully centralized.

4 Performance Improvements Evaluation

We carried out a series of experiments to assess the performance improvements induced by the use of our enhanced `MPI_Dist_graph_create` function. All tests are executed on a cluster composed of 68 nodes linked with an Infiniband interconnect (HCA: Mellanox Technologies, MT26428 ConnectX IB QDR). Each node is composed of two Intel Xeon Nehalem X5550 cpus featuring 4 2.66 GHz cores each. The 8 Mbytes of L3 cache are shared between the four cores of a CPU. There are 24 GB of DDR3 RAM at 1.33 GHz on each node. As for the software, the operating system is SLES 11 and the MPI implementation is MVAPICH2 1.7 (alpha 1) [10]. All of the benchmarks involve 64 processes (8 nodes connected to the same Infiniband switch are used) and each process is bound to its dedicated core. The baseline chosen to compare the process placement policies is the *Serial Ranking* policy where the process rank number n (in `MPI_COMM_WORLD`) is placed on the node number m with $m = n/8$ ($n \in [0, 63]$ in our case). Such a policy is enforced by default by most resource schedulers when providing a machine file to the user after reserving nodes (e.g PBS/Torque). Also, the execution times do not take into account the time spent in the `MPI_Dist_graph_create` function called at the begining of each benchmark (less than 140 milliseconds in our experiments with 64 processes). The tests are run several times in a row: the *Serial Ranking* case (without reordering) is followed by the reordered cases, using the two metrics listed in Sec. 3.2.



(a) Absolute

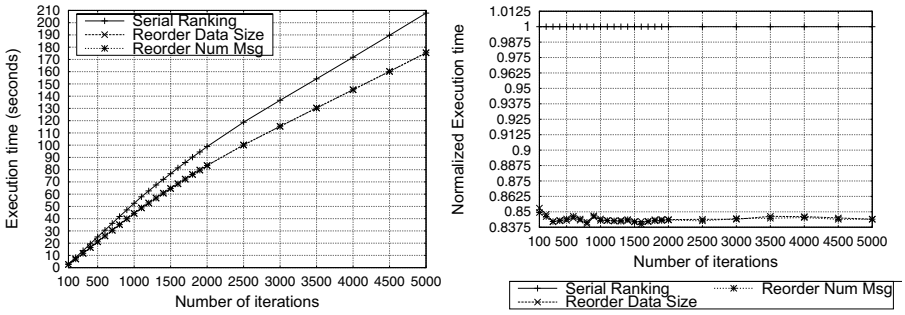
(b) Normalized (Round Robin is not shown due to its poor performance)

Fig. 1. Ring pattern execution times

This first execution actually creates the communication pattern file we need to initialize the pointer arrays of the `MPI_Dist_graph_create` function called in the following runs.

4.1 The Ring Pattern Benchmark

The first benchmark is purposely designed to showcase the benefits of the reordering technique. The communication pattern features several rings of processes of equal sizes. A token is exchanged in each ring, that stops circulating when received back by the process that initially sent it (a.k.a the *ring leader*). Then, all ring leaders exchange the token with a call to `MPI_Allgather`. The test is run with 8 rings composed of 8 processes each. For this test, we make experiments with two non-reordered cases: the first one is when the *Serial Ranking* policy (as described above) is used and the second one is when a so-called *Round Robin* policy is used. With this policy, core number i of node number j executes process rank n (in `MPI_COMM_WORLD`), where: $n = (8 * i) + j$ and $(i, j) \in [0, 7]^2$ (cores and nodes are numbered linearly). The process-to-core binding policy within nodes is the same for both policies. We use the Round Robin case in order to show that a suboptimal process mapping/binding policy can be effectively corrected by reordering. In our example, since 8 nodes of the cluster are used and given the communication pattern, most of the traffic goes through the network in the Round Robin case. In the reordered cases, most of communication uses shared memory, much faster than the network. Indeed, Figure 1 shows that our algorithm is able to compute a relevant reordering and that we are able to exploit the nodes internal structure more finely since the execution times achieved by either the *Data Size* or *Number of Messages* cases are 10% to 20% faster than in the *Serial Ranking* case. Since the amount of data and the number of messages grow proportionaly, both metrics yield the same results.



(a) Absolute (b) Normalized

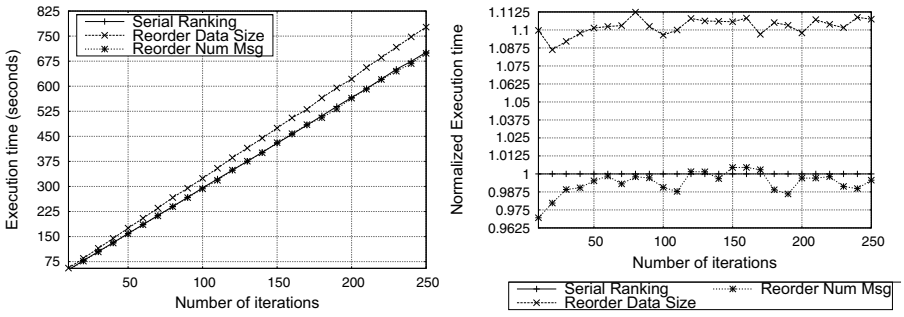
Fig. 2. ZEUS-MP Execution times

4.2 ZEUS-MP

The second set of experiments involve a real application called ZEUS-MP. It is a computational fluid dynamics code for the simulation of astrophysical phenomena that solves magnetohydrodynamics equations. The three-dimensional computational domain is organized in tiles where the boundary data is exchanged with MPI messages between neighbours. We used the 2.1.2 [11] version of ZEUS-MP. Originally, this application uses the MPI cartesian topology mechanism, but without reordering. Also, ZEUS-MP is not able to take into account the underlying physical architecture thanks to options or arguments passed to the program for instance. We ran ZEUS-MP for various iteration counts and measured the execution times. Figure 2 shows the benefits of using reordering. Since our modified version of `MPI_Dist_graph_create` allow the application to better exploit the underlying multicore architecture, the cost of communication is reduced. The overall execution times are decreased by more than 15%. Both metrics yield the same results. This results shows that our approach is relevant and allows the user to better exploit the nodes internal structure, without possessing a prior knowledge of their physical topology. To manage to get equivalent results, the programmer would have to: 1– understand the application behaviour, leading to the use the *Serial Ranking* policy and 2– provide an adequate process-to-core binding when running it.

4.3 RSA-768 – The Block Wiedemann Algorithm

The 768-bits, 232-digits number RSA-768 is factorized since December 12, 2009 [12]. Several algorithms and applications were used to achieve this result and one particular step consists to find dependencies between the rows of a sparse matrix using a *Block Wiedemann* algorithm. We benchmarked a simplified version of this Block Wiedemann step provided by one of the authors of [12]. It is



(a) Absolute (b) Normalized

Fig. 3. Block Wiedemann step – RSA-768 Execution times

a relevant target for our work because it is a *communication bound* application. This application takes into account the underlying physical architecture thanks to parameters passed to the program (i.e the number and layout of the cores). This is a difference with our previous application case (ZEUS-MP). This version of *Block Wiedemann* is designed to be used with the *Serial Ranking* placement policy. Figure 3 shows the results obtained. Out of the two metrics, only *Number of Messages* manages to slightly improve the results obtained with *Serial Ranking*. The execution times are decreased by less than 2%. This results demonstrate that some room for improvement exists even for applications that are tuned to exploit the underlying architecture. All these results therefore advocate for reordering, and demonstrate how this technique can alleviate some of the burden of tuning an MPI application to an underlying multicore system. Thanks to `MPI_Dist_graph_create`, this tuning is performed transparently, automatically and in a portable fashion.

5 Related Works

The placement of MPI processes on processors in order to match the communication pattern to the underlying hardware architecture has been previously examined. The problem is introduced in [1] and an algorithm, based on the Kernighan-Lin heuristic [13], is described as well as results for some benchmarks. However, this work is tailored for a specific vendor hardware and is thus not suitable for a generic case. Also, the author optimizes some of the routines creating *cartesian topologies* but left unaddressed the more generic *graph topology* case. The experiments show dramatic improvements, but are restricted to benchmarks only communicating but not doing any computation. In particular, the *Jacobi method for a Poisson problem solver test* results are consistent with of our own ring test results. Topology mechanisms implementation issues are also discussed in [2]. Both cartesian and graph topologies are addressed by this work, and the

algorithm is based on the same Kernighan-Lin heuristic. The optimization criterion considered is either the total communication cost (as in the TreeMatch algorithm) or the optimal load balance. But here again, it is a work designed for a specific vendor hardware. The approach is thus less generic than our. Besides the Kernighan-Lin and TreeMatch, there are other algorithms that can solve the *graph embedding problem*. Scotch [14] is a graph coupling framework but not optimized for our case where we work only on *trees* (see Sec. 3.3). A previous version of our work, which concerned the core binding technique, did in fact use Scotch [5]. TreeMatch, however, outperforms Scotch in terms of execution times and is therefore better suited for runtime optimizations. MPIPP [15] is another framework aiming at optimizing an application execution on the underlying hardware. MPIPP relies on an external tool to generate the hardware information while we manage to perform this at runtime. A comparison between TreeMatch and MPIPP can be found in [9]. Also, the MPIPP framework uses the core binding technique, as well as a couple of vendors such as Cray [16], HP [17] and probably IBM [18].

6 Conclusion and Future Works

In this paper, we showed that using rank reordering can allow MPI applications to transparently exploit clusters of multicore nodes. The application communication pattern is matched to the underlying hardware, thus reducing the cost of application communication. The communication pattern is usually expressed as the overall amount of bytes exchanged among processes but we experienced that other metrics are more relevant in our particular environment, that is using MVAPICH2 as the MPI implementation and TreeMatch as the algorithm to solve the *graph embedding problem*. Indeed, using the number of messages to characterize the communication pattern yields better results than the amount of bytes. We plan to understand why the *Number of Messages* metric outperforms in some cases the *Data Size* one. Also, our `MPI_Dist_graph_create` implementation is currently centralized, which is not scalable. We would like to implement a distributed version, but this might imply to distribute the TreeMatch algorithm itself. This algorithm could also integrate new optimization criteria such as the ones listed in [6]. Currently, we do not take into account the physical topology of the network. We only exploit the nodes internal structure. There are plans to expand Hwloc in order to provide such information, we are hence looking forward to take advantage of it. Also, we currently lack some quantitative information about NUMA effects. Indeed, a more recent release of Hwloc (1.2) features latency matrices in order to assess performance between cores. It is another piece of information that we want to exploit. At last, we plan to work on the extraction of the communication pattern information without relying on a previous run of the complete application. A static analysis of the MPI code could be performed at compile time to generate the needed information.

References

1. Hatazaki, T.: Rank reordering strategy for MPI topology creation functions. In: Alexandrov, V.N., Dongarra, J. (eds.) PVM/MPI 1998. LNCS, vol. 1497, pp. 188–195. Springer, Heidelberg (1998)
2. Träff, J.L.: Implementing the MPI process topology mechanism. In: Supercomputing 2002: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing, pp. 1–14. IEEE Computer Society Press, Los Alamitos (2002)
3. Zhu, H., Goodell, D., Gropp, W., Thakur, R.: Hierarchical Collectives in MPICH2. In: Ropo, M., Westerholm, J., Dongarra, J. (eds.) PVM/MPI. LNCS, vol. 5759, pp. 325–326. Springer, Heidelberg (2009)
4. Ma, C., Teo, Y.M., March, V., Xiong, N., Pop, I.R., He, Y.X., See, S.: An Approach for Matching Communication Patterns in Parallels Applications. In: Proceedings of 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2009). IEEE Computer Society Press, Rome (2009)
5. Mercier, G., Clet-Ortega, J.: Towards an efficient process placement policy for MPI applications in multicore environments. In: Ropo, M., Westerholm, J., Dongarra, J. (eds.) PVM/MPI. LNCS, vol. 5759, pp. 104–115. Springer, Heidelberg (2009)
6. Hoefler, T., Rabenseifner, R., Ritzdorf, H., de Supinski, B.R., Thakur, R., Träff, J.L.: The scalable process topology interface of mpi 2.2. Concurrency and Computation: Practice and Experience 23, 293–310 (2011)
7. Argonne National Laboratory: MPICH2 (2004), `http://www.mcs.anl.gov/mpi/`
8. Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., Namyst, R.: 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 (PDP 2010), Pisa, Italia. IEEE Computer Society Press, Los Alamitos (2010)
9. Jeannot, E., Mercier, G.: Near-optimal placement of MPI processes on hierarchical NUMA architectures. In: D'Ambra, P., Guarracino, M., Talia, D. (eds.) Euro-Par 2010. LNCS, vol. 6272, pp. 199–210. Springer, Heidelberg (2010)
10. Design of High Performance MVAPICH2: MPI2 over InfiniBand. In: Proceedings of the 6th IEEE International Symposium on Cluster Computing and the Grid. IEEE Computer Society, Los Alamitos (2006)
11. Hayes, J.C., Norman, M.L., Fiedler, R.A., Bordner, J.O., Li, P.S., Clark, S.E., Ud-Doula, A., Mac Low, M.-M.: Simulating Radiating and Magnetized Flows in Multiple Dimensions with ZEUS-MP. The Astrophysical Journal Supplement 165, 188–228 (2006)
12. Kleinjung, T., Aoki, K., Franke, J., Lenstra, A.K., Thomé, E., Bos, J.W., Gaudry, P., Kruppa, A., Montgomery, P.L., Osvik, D.A., te Riele, H., Timofeev, A., Zimmermann, P.: Factorization of a 768-bit RSA modulus. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 333–350. Springer, Heidelberg (2010), `http://www.springerlink.com`
13. Kernighan, B.W., Lin, S.: An efficient heuristic procedure for partitioning graphs. Bell System Technical Journal 49, 291–307 (1970)
14. Pellegrini, F.: Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs. In: IEEE Proceedings of SHPCC 1994, Knoxville, pp. 486–493 (1994)

15. Chen, H., Chen, W., Huang, J., Robert, B., Kuhn, H.: Mpipp: an automatic profile-guided parallel process placement toolset for smp clusters and multiclusters. In: Egan, G.K., Muraoka, Y. (eds.) ICS, ACM, pp. 353–360 (2006)
16. National Institute for Computational Sciences: (MPI Tips on Cray XT5), http://www.nics.tennessee.edu/user-support/mpi-tips-for-cray-xt5
17. Solt, D.: A profile based approach for topology aware MPI rank placement (2007), http://www.tlc2.uh.edu/hpcc07/Schedule/speakers/hpcc_hp-mpi_solt.ppt
18. Duesterwald, E., Wisniewski, R.W., Sweeney, P.F., Cascaval, G., Smith, S.E.: Method and System for Optimizing Communication in MPI Programs for an Execution Environment (2008), http://www.faqs.org/patents/app/20080288957