An Overview of Process Mapping Techniques and Algorithms in High-Performance Computing

Torsten Hoefler, Emmanuel Jeannot, Guillaume Mercier

CHAPTER 5

AN OVERVIEW OF TOPOLOGY MAPPING ALGORITHMS AND TECHNIQUES IN HIGH-PERFORMANCE COMPUTING

TORSTEN HOEFLER[1], EMMANUEL JEANNOT[2], AND GUILLAUME MERCIER[3,2]

[1]ETH Zurich, Switzerland
[2]Inria Bordeaux Sud-Ouest, Talence, France
[3]Bordeaux Polytechnic Institute, Talence, France

Due to the advent of modern hardware architectures of high-performance computers, the way the parallel applications are laid out is of paramount importance for performance. This chapter surveys several techniques and algorithms that efficiently address this issue: the mapping of the application's virtual topology (for instance its communication pattern) onto the physical topology. Using such strategy enables to improve the application overall execution time significantly. The chapter concludes by listing a series of open issues and problems.

5.1 Introduction

High Performance Computing (HPC) applications are increasingly demanding in terms of computing power. Currently, this computing power can be delivered by

High-Performance Computing on Complex Environments, First edition.
By Edited by Emmanuel Jeannot and Julius Žilinskas.

parallel computers only. At the same time, the trend in processors and parallel architecture design is to increase the number of computing cores. Therefore, exploiting efficiently such a complex hardware, that is, bridging the performance gap between the target machine level and the application level is a challenging issue. Moreover, current and future generations of applications will have to tackle this challenge in order to scale because of the expected increase concurrency in the applications and the input data size. As a consequence, the way the application data are organized, accessed and moved is of paramount importance and should be improved and optimized. This *Locality* issue exists at several levels: besides the application data layout, locality issues also stem from the use of multicore nodes (which feature a hierarchical memory organization), network and storage units. Since the time to transfer data between *processing entities* (e.g., processes or threads) of the application depends on both, the affinity of these entities *and* their respective locations, a thorough analysis of the application behavior (data-wise) and of the platform on which it is executed is necessary. Given both pieces of information, clever algorithms and techniques can dramatically improve the application communication time by carefully mapping the application processing entities on the various processing units (e.g., CPUs or cores) of the architecture. The goal of this chapter is to provide an overview of this *topology mapping* issue and to present the various techniques and solutions existing in the literature. This chapter is divided into the following six parts: first we present an overview of the problem in Sect. 5.2. Then, in Sect. 5.3, we show a possible formalization of the problem. Section 5.4 presents existing algorithms that can yield solutions to the topology mapping problem while Sect. 5.5 exposes the various techniques that applications can leverage to enforce the mapping computed by the algorithm. Section 5.6 describes existing software implementing the algorithms and the techniques seen in the previous sections. Last, in Sect. 5.7, we present a set of open problems and research issues that need to be addressed in order to pave the road to Exascale.

5.2 General Overview

High-Performance Computing (HPC) is more than ever a cornerstone of the development and competitiveness of modern and knowledge-based societies and economies. In both fields of science and technology, it is necessary to solve problems requiring tremendous amounts of computing power, memory and storage. From an industrial perspective, parallel computers are mandatory to reduce the time-to-market of a large array of products (e.g., cars, drugs, planes, etc.) whilst from a research perspective, simulations require to refine models in order to solve larger problems at longer and finer time-scales. Therefore, many scientific domains have already been identified as in need for large amounts of computational power, as for instance by the PRACE research infrastructure[1], which has selected the five domains: (1) Weather, Climatology and solid Earth Sciences; (2) Astrophysics, High-Energy Physics and Plasma Physics; (3) Materials Science, Chemistry and Nanoscience; (4) Life Sciences, Medicine and (5) Engineering Sciences and Industrial Applications. Paral-

lel architectures are the only ones able to deliver the much sought-after computing power.

5.2.1 A Key to Scalability: Data Locality

However, to harness the resources of a parallel computer is by no means a trivial undertaking since multiple challenges need to be addressed, one of the hardest being to jump on the multicore/manycore bandwagon: the size of the machines in terms of number of cores per node is increasing. Exascale computers are expected to feature between hundreds of thousands to millions of nodes, each them integrating between a hundred to ten thousands of cores. Dealing with such large scales is very challenging. Current generations of machines are already hierarchical, both in terms of network interconnection and memory architecture (cache level, non-uniform access, coherency, etc.) and future generations of machines will feature even deeper hierarchies. As a consequence, the challenge deals with scalability and can be expressed in several ways: how to use the maximum of the available resources (e.g., CPUs, cores, memory, network, etc.) at their full potential? How to do so with an energy consumption that remains (economically and environmentally) acceptable? One global and practical answer to these questions is to improve the *Data Locality* of parallel applications, that is, the way the data are placed, accessed and moved by the multiple hardware processing units of the underlying target architecture.

This is coherent with the behavior of a parallel application that dispatches its workload among *software processing entities* (e.g., tasks, processes, threads) running in parallel on the *hardware processing units* of the target architecture. These processing entities access and exchange data during the application's execution but not necessarily in a regular fashion. As a consequence, these data accesses and exchanges can be optimized to fully exploit the hardware. For instance, entities exchanging or sharing lots of data could be placed on hardware processing units physically close one to the other. By doing so, the communication costs are reduced, thus decreasing the application's overall execution time and as a consequence its energy consumption, as emphasized by the IESP roadmap [2]: "*Since much of the power in an Exascale system will be expended moving data, both locally between processors and memory as well as globally, the X-stack must provide mechanisms and APIs for expressing and managing data locality. These will also help minimize the latency of data accesses.*"

5.2.2 Data Locality Management in Parallel Programming Models

One other possible answer to the scalability issue would be to use a new parallel programming paradigm. However, currently, no existing paradigm seems viable, as parallel application developers still rely on proven and wide-spread tools created years ago. This fact is a consequence of a software inertia as well as the huge number of existing legacy applications.

Nevertheless, data locality has to be taken into account to improve scalability of present and future parallel applications, regardless of the chosen paradigm. Hence,

current existing paradigms can be improved/enhanced to better express this locality management. They indeed offer a simplified view of the architecture: the Message Passing Interface (MPI) relies on a distributed memory model with a flat network whilst OpenMP uses a global address space and Partitioned Global Address Space (PGAS) languages use similarly to MPI a flat, partitioned address space. As a consequence, the gap between the underlying hardware and the simple view exposed by the programming model widens. To bridge this gap, paradigm implementations should first be optimized to better reflect the locality management and second, these paradigms' definition and utilization (locality-wise) should be more specific.

More precisely, the necessary interactions between the applications (relying on paradigm implementations) and the underlying hardware should be explicitly defined and expressed in a synthetic fashion. Improving an application's data locality may carry different results, depending on the context and on the programming paradigm used. For instance, in the realm of message passing-based applications, it may lead to a decrease of the overall communication costs whilst for applications based on multithreading, the expected results shall be a better sharing of the data between the application threads. This implies to better use the underlying hardware: the network, the memory hierarchy and of course the processing units available.

5.2.3 Virtual Topology: Definition and Characteristics

A simple, yet efficient way to improve data locality is to dedicate physical processing units to their specific software processing entities. This means that a *matching* between the application virtual topology and the target hardware architecture has to be determined. In this chapter, the expression of *virtual topology* designates a mean to express the existing *dependencies between software processing entities*. For instance, in programming paradigms with explicit communications (e.g., as in message passing), these dependencies are expressed as messages exchanged between processes whilst for implicit communications paradigms (e.g., OpenMP) these dependencies are expressed as accesses to common memory locations by the application threads. This virtual topology representation is also tailored for task-based programming environments as the various tasks depend one from the others: indeed, a task can only be scheduled once a set of "previous" tasks is completely executed. Virtual topologies are also often referred to as application *communication patterns*.

Such communication pattern can be either *static* or *dynamic*. Static means that the number of processing entities remains constant during the course of the application and that the dependencies between these entities do not change between consecutive application steps. The pattern can be qualified as dynamic when one of the two above conditions (possibly both) is not fulfilled. For instance, an OpenMP-based application features a dynamic pattern since new threads are created/destroyed when entering/exiting each new parallel section. This communication pattern (or virtual topology) can be considered as a key characteristic of the application [3].

5.2.4 Understanding the Hardware

If, on one hand, understanding the behavior of the application is necessary, on the other hand, the maximum of details regarding the target hardware have to be gathered. Achieving this in a convenient and portable fashion is of paramount importance in order to address the largest spectrum possible of parallel architectures. This issue is increasingly complex due to the advent of multicore/manycore CPUs. Indeed, parallel computers used to follow the cluster paradigm that possessed an architectural complexity only at the network level. But multicore CPUs pushed the envelope further because of their intricate and hierarchical memory organization, inducing NUMA effects. Performance of data accesses and movements between processing units is now heterogeneous. As a consequence, both the physical network topology *and* the multicore node internal structure have to be considered in order to efficiently determine the matching between the virtual and the physical topologies.

As a matter of fact, this matching between the virtual and the physical topologies is achievable in both ways: the virtual topology can be mapped onto the physical one, but the physical topology can also be mapped onto the virtual one. The first case corresponds to a resource allocation problem where the software processing entities have to be mapped onto their hardware processing counterparts. This problem is usually referred to as a *process mapping* issue. In the second case, the hardware can be tailored to fit application structure (virtual topology). This is feasible with software-reconfigurable networks, for instance.

Hence, the key is to make use of an algorithm/heuristic that yields a satisfactory solution to our topology mapping problem. In the following section we outline the main algorithms used to compute such mapping as well as environments that application developers can use to apply this mapping.

5.3 Formalization of the Problem

Abstractly seen, the topology mapping problem can be phrased as a minimization problem of various metrics. We will now discuss the static topology mapping problem and later generalize to a dynamic version that is relevant for task-based environments.

The network is typically modeled by a weighted graph $H = (V_H, \omega_H, \mathcal{R}_H)$ where the set of vertices, $V_H \in \mathbb{N}$, represents the execution units and the weighted edges $\omega_H(u, v) \in \mathbb{R}$ with $u, v \in V_H$ represent the weight of the edge between the two vertices u and v. Non-existing edges can be modeled by the weight zero. The function $\mathcal{R}_H(u, v)$ represents the routing as a probability distribution on the set of simple paths $P(u, v)$ between vertices u and v. Various previous works choose to model the network as a specialized subset of this abstract specification (for example, the routing function $\mathcal{R}_H$ is most commonly ignored and the weights $\omega_H(u, v)$ are often replaced by binary values to indicate the existence of an unweighted edge).

The static application graph is often modeled as a weighted graph $A = (V_A, \omega_A)$, where V_A represents the set of communicating processes and $\omega_A(u, v)$ some metric

for the communication between two processes $u, v \in V_A$. There is no general agreement on how application communication is modeled. Some works propose the total message volume of a certain phase of communication, others propose message size, or the number of messages as a model for application communication. Again, almost all previous models fit our general specification.

The topology mapping now considers mappings $\sigma : V_A \to V_H$, i.e., σ assigns each vertex $s \in V_A$ in the application graph a target vertex $t \in V_H$ in the architecture (host) graph. Some works assume σ to be injective or surjective, however, in the general case, it has to be neither (it may map multiple vertices in V_A to the same target vertex in V_H and it may also leave target vertices in V_H unassigned).

Each concrete mapping σ has an associated cost metric, which is typically the target of the optimization (often minimization) problem. As for the communication metrics, there is no general agreement. We distinguish two fundamentally different metrics: dilation and congestion. Informally, dilation is defined as either the maximum or the sum of the pairwise distances of neighbors in A mapped to H. For example, let $d_H(x, y)$ be the shortest distance between vertices $x, y \in V_H$, the weighted sum of the dilation is defined as

$$\sum_{u,v \in V_A} d_H(\sigma(u), \sigma(v)) \times \omega(u, v).$$

We note that the routing function $\mathcal{R}_H$ can increase the dilation if routes are not along shortest paths. Thus, an algorithm that includes the routing function may find more practical process mappings. The sum (or average) dilation allows a comparison of the number of times packets are transmitted over network interfaces, Thus, this metric often correlates strongly with the dynamic energy consumption of the network.

A second fundamental metric is the congestion, which counts how many communication pairs use a certain link. Here, it is necessary to define a routing of messages. However, if $\mathcal{R}_H(u, v)$ is not specified or known, one can always use shortest path routing between u and v (i.e., all edges on a single shortest path between u and v have routing probability 1 while all other edges have probability 0). Let $p_e(u, v)$ be the probability that any of the routes from u to v crosses an edge $e \in V_H$. Then, we can define the congestion of this edge e as:

$$C_e = \sum_{u,v \in V_A} p_e(u, v).$$

Again, we can define various reduction functions to generate an overall measure of congestion across all network links. Most common is a measure for the maximum congestion

$$C_{\max} = \max_e C_e,$$

which often correlates strongly with the execution time of bulk-synchronous parallel (BSP) applications.

Optimization algorithms would now strive to minimize any of those metrics in our presented model or a subset thereof. For example, if one was to find a mapping that

optimizes dynamic energy consumption, one would strive to minimize dilation while one would minimize the maximum congestion in order to optimize the completion time of a BSP application.

A simple generalization can be made for tasking systems. Here, the graph A is the data-dependency graph of all active tasks in the system. Static optimizations can be performed as before (we ignore the orthogonal load balancing problem in this regard). If a new task is to be spawned, one could either solve the whole mapping problem from the start with an updated A' or one could place the task heuristically "close" to its neighbors.

5.4 Algorithmic Strategies for Topology Mapping

It can be shown that most specific problems of mapping arbitrary A to arbitrary H with regards to any of the metrics are NP-hard. In fact, many of the generic optimization problems can be phrased as a quadratic assignment problem, which belongs, even for highly structured inputs, to the class of strongly NP-hard problems. This means that the problem cannot be approximated well. Using today's solvers, quadratic assignment problems may be solved for very small input instances, but are generally unpractical for data centers or large-scale computing networks.

On the other hand, certain idealized mapping cases can be solved in polynomial time. For example, embedding a $k \times l$ cube (2D Torus) into another $k \times l$ cube is simple, or embedding a line of length l into the same cube is simple as well. Numerous works have studied such special and idealized mappings in the past. However, such ideal structures are rarely found in reality. Thus, and due to space reasons, we limit our report to the mapping of arbitrary A to arbitrary H (with some relevant exceptions, such as k-ary n-cubes).

Various heuristics have been developed in the past. This section provides a quick overview of the methods. References to specific works using those methods are provided later. New schemes are invented continuously making this list a quickly moving target. Such schemes may or may not fit into this classification, however, we believe that those classes cover most of the existing works in the field of topology mapping. Each of those schemes performs well for a certain class of graphs. An exact classification and identification of such graph classes is subject of ongoing research.

5.4.1 Greedy Algorithm Variants

The probably simplest schemes for mapping are derived from well-known greedy strategies. For example, in a local greedy scheme, one selects two starting vertices $u \in V_H$ and $v \in V_A$, and adds other vertices to the mappings walking along the neighborhood of both vertices. A global greedy scheme would greedily select the next vertex based on some global property, i.e., the weight of all out-edges. One can also mix local and global schemes on the two graphs A and H.

5.4.2 Graph Partitioning

A second general scheme that is often used for topology mapping is graph partitioning. For example k-way partitioning or it's special case bipartitioning (i.e., when k equals 2) can be used to recursively cut both graphs (A and H) into smaller pieces, which are then mapped while unfolding the recursion. A well-known heuristic for bipartitioning is Kernighan-Lin [4].

5.4.3 Schemes Based on Graph Similarity

Another class of schemes, first explored in [5], is based on graph similarity. Here, the adjacency lists of the two graphs are permuted into some canonical form (e.g., minimizing the bandwidth of the adjacency matrix using well known heuristics) such that the two graphs can be mapped based on this similarity.

5.4.4 Schemes Based on Subgraph Isomorphism

Some schemes base on subgraph isomorphism. For this, we assume that H has more vertices than A and that we try to find a set of target vertices in H to map A to. Several fast algorithms exist for approximating this problem.

5.5 Mapping Enforcement Techniques

In this section, we detail the various techniques that application programmers can use in order to enforce the mapping computed by the dedicated algorithms described in the previous section. We remind that this computation is the outcome of a three-step process:

1. The virtual topology (communication pattern) of the target application is gathered.
2. The physical topology of the target underlying architecture is gathered (or modeled).
3. The matching between both topologies is computed thanks to the relevant algorithm/heuristic, and then applied.

It is worth to note that the question of how both pieces of information regarding the topologies are gathered (i.e., the first two aforementioned steps) is out of the scope of this survey that focuses only on the last step. As explained previously, mapping the virtual topology onto the physical one can be achieved by determining the number of the assigned physical processing unit for each of the application's processing elements. As a consequence, enforcing such a mapping comes down to applying a certain *placement policy* for the considered application. In the remainder of this section, we give details about the various techniques that allow a programmer to apply such a placement policy.

5.5.1 Resource Binding

In order to apply the relevant placement policy, one first obvious technique is to bind the processing elements to their dedicated hardware processing units. For instance, in a GNU/Linux-based system, commands such as `numactl` or `taskset` fulfill this goal. However, there is no portable solution available across a wide spectrum of systems and architectures. The Hardware Locality tool (Hwloc) [6] partly solves this problem by providing a generic, system-independent interface that exposes and abstracts the underlying memory hierarchy and processors layout in order to manage the placement of processing elements. Binding a processing element to its hardware unit is, to some extent, a way of regaining control over the scheduling policy of the operating system. As a matter of fact, when no binding is enforced, the operating system scheduler can swap any processing entity to any processing unit, thus leading to cache misses that may harm performance. Moreover, as the scheduling of processes/threads is not deterministic, the impact on the performance may vary from one run to an other: application performance is thus less predictable than in the case of bound entities. For instance, in Table 5.1, we compare the same execution of a CFD application (ZEUS-MP), when either processes are bound to cores or not. We show the mean execution time, the standard deviation and the coefficient of variation – CV (the standard deviation normalized by the mean) for ten runs and different numbers of iterations. As the means are not equal, only the CV is significant, we include all data for completeness. We see that in all cases, the CV is lower for the binding case than for the non-binding case, meaning that binding processes to cores leads to decreased system noise and more stable execution times.

Table 5.1: Statistics for ten runs of ZEUS-MP/2 CFD application with 64 processes (MHD blast case) comparing the binding case and the non binding case (Courtesy of Jeannot and Mercier).

Number of	No Binding			Binding		
Iterations	Mean	Std. Dev.	Coef. of Var.	Mean	Std. Dev.	Coef. of Var.
2000	2.8627	0.127	0.044	2.6807	0.062	0.023
3000	4.1691	0.112	0.027	4.0023	0.097	0.024
4000	5.4724	0.069	0.013	5.2588	0.052	0.010
5000	7.3187	0.289	0.039	6.8539	0.121	0.018
10000	13.9583	0.487	0.035	13.3502	0.194	0.015
15000	20.4699	0.240	0.012	19.8752	0.154	0.008
20000	27.0855	0.374	0.014	26.3821	0.133	0.005
25000	33.7065	0.597	0.018	32.8058	0.247	0.008
30000	40.6487	0.744	0.018	39.295	0.259	0.007
35000	46.7287	0.780	0.017	45.7408	0.299	0.007
40000	53.3307	0.687	0.013	52.2164	0.227	0.004
45000	59.9491	0.776	0.013	58.8243	0.632	0.011
50000	66.6387	1.095	0.016	65.3615	0.463	0.007

Since it usually relies on commands that are outside of the programming paradigm itself (e.g., the process manager in MPI implementations), binding can be performed without any application modifications. However, this induces two issues: first, portability is not guaranteed, as commands may vary from one system to the other. Second, changing this binding during the course of the application execution can be difficult to achieve in a standard fashion.

5.5.2 Rank Reordering

Another technique to enforce the placement policy determined by the matching algorithm is called *rank reordering* [7],[8]. Each processing entity of the parallel application possesses its own identification number. This number can be used, for instance, to exchange data or to synchronize the entities. The rank reordering technique allows to modify these identification numbers, so as to better reflect the application's virtual topology. Rank reordering does often not rely on external commands/tools of the system and may be part of the programming standard itself (e.g., in MPI). Therefore, legacy applications have to be modified to take advantage of this technique, the scope of theses changes varying from one paradigm to the other. However, relying on a standard feature ensures portability, transparency and dynamicity since it can be issued multiple times during an application execution.

However, reordering the ranks is not by itself a sufficient mean to improve application performance. Indeed, side-effects of poor scheduling decisions (cache misses, etc.) can still apply to applications using only rank reordering. That is why a joint use of resource binding and rank reordering is the most sensible combination of techniques to apply the placement policy. This can be achieved in a two-step process: first, processing entities are bound to processing units when the application is launched. For this step, there is no relevant placement policy to apply, since this binding is enforced just to avoid the scheduling side-effects. Then, in a second phase (and during the application execution itself), the ranks of the processing entities are effectively reordered according to the results yielded by the matching algorithm.

5.5.3 Other Techniques

Resource binding and rank reordering are the most prevalent schemes in high-performance computing. However, other fields, such as Operating Systems and Distributed Systems may use different mechanisms. For instance, an operating system may observe memory traffic in a NUMA node and move processes closer to their respective memory banks in order to minimize cross-link traffic [9]. Another example would be optimized network placement algorithms [10]. However, a detailed explanation of such techniques, outside the realm of high-performance computing, is beyond the scope of this survey.

5.6 Survey of Solutions

In this section, we provide an overview of work related to the generic topology mapping problem. As mentioned earlier, we have to omit many specialized solutions (e.g., for certain graph classes) for space reasons. However, we aim at covering all generic topology mapping schemes at a rather high level and refer to the original publications for details.

We classify each work as either a purely algorithmic solution or as a software implementation (which may include algorithmic work as well). Works that fall in both categories default to the software solution section.

5.6.1 Algorithmic Solutions

The topology mapping problem is often modeled as a *Graph Embedding Problem*: one formulation of the embedding problem is introduced by Hatazaki [11] while Rosenberg [12] discusses the complexity and an algorithm for the embedding problem.

Bokhari [13] models the mapping problem as a graph isomorphism problem. However, the strategy described ignores edges that are not mapped. It was shown later that such edges can have a detrimental effect on the congestion and dilation of the mapping. Lee and Aggarwal [14] improve those results and defines a more accurate model which includes all edges of the communication graph and propose a two-stage optimization function consisting of initial greedy assignment and later pairwise swaps. Bollinger and Midkiff [15] use a similar model and simulated annealing to optimize topology mappings.

Sudheer and Srinivasan [16] model the optimization problem for minimizing the weighted average dilation metric (called hop-byte) as a quadratic assignment problem. However, the conclusion is that only very small instances can be solved by this approach. A heuristic to minimize the average hop distance is proposed.

Many practical schemes that will be described in the next section use recursive partitioning (or bisection as a special case) for topology mapping. However, Simon and Teng [17] show that recursive bisection does not always lead to the best partitions.

5.6.2 Existing Implementations

In this section, we describe existing software packages that can be used to approach the topology mapping problem. We start with graph partitioning software that can be employed in conjunction with the recursive partitioning schemes. Then, we discuss specialized solutions and analyses for various network topologies, followed by a description of generic mapping software. Finally, we discuss support for topology mapping in current parallel programming frameworks.

5.6.2.1 Graph Partitioning Software We now list some graph partitioning software packages. The typical use-case for those packages is the partitioning of large

graphs for the parallelization of scientific computing problems. The heuristics used in those packages may thus not always be suitable for partitioning small graphs.

Metis [18] and its parallel version ParMetis [19] are among the most used well established graph partitioners. The Chaco [20] and Zoltan [21] graph partitioners maintained by Sandia National Laboratories employs a variety of different partitioning schemes. SCOTCH [22] is a graph partitioning framework able to deal with tree-structured input data (called *tleaf*) to perform the mapping. Scotch is based on dual recursive bipartitioning. Other graph partitioners, such as Jostle [23], are available but less commonly used in larger software packages.

5.6.2.2 Mapping for Specific Topologies The mapping problem is often studied in the context of particular network topologies or technologies. We proceed to give a quick overview of current network technologies: Torus networks are used in different variations in IBM's Blue Gene series (BG/L, BG/P [24] and BG/Q [25]), Cray's Gemini network [26], and Fujitsu's K computer [27]. A second large class of topologies is the family of fat tree networks [28, 29, 30] which is often used in commodity datacenters and high performance interconnects. Fat trees usually offer higher bisection bandwidth than torus networks. The Dragonfly topology [31] and variants are used in IBM's PERCS system [31, 32] and Cray's Aries network [33] and promises high bisection bandwidth at lower costs. Those three classes of topologies form the base of most of today's supercomputer networks. Thus, topology mapping schemes should aim at supporting those topologies.

Some research works thus address only generic application topologies (used to express the communication pattern) but consider only the network physical topology for the hardware aspects. Balaji et al. [34], Smith and Bode [35], and Yu et al. [36] provide mapping strategies and software for Blue Gene systems as target architectures. Subramoni et al.[37] discuss how to map processes on InfiniBand networks. Other works, by Rashti et al. [38], Träff [39], and Ito et al. [40] target specifically multicore networks.

Von Alfthan et al. [41] target several classes of architectures such as the Cray XT, the BlueGene/P and the generic multicore networks. They use a custom, non-standard interface to build the graph representing the topologies, despite the presence of this functionality in MPI. The network topology is gathered dynamically in the case of Cray and IBM hardware, but is considered flat in case of the generic multicore network.

TREEMATCH ([42],[43]) is an algorithm and a library for performing topology-aware process placement. Its main target are networks of multicore NUMA nodes. It provides a permutation of the processes to the processors/cores in order to minimize the communication cost of the application. It assumes that the topology is a tree and does not require valuation of the topology (e.g. communication speeds). Therefore, TREEMATCH solution is based only on the structure of the application and the topology and is therefore independent from the way communication speeds are assessed. TREEMATCH also implements different placement algorithms that are switched according to the input size in order to provide a fast execution time, allowing dynamic load-balancing for instance.

5.6.2.3 Mapping Frameworks/Libraries LibTopoMap [5] is a generic mapping library that implements MPI-2.2's topology interface [44] using various heuristics such as recursive bisection, k-way partitioning, simple greedy strategies, and Cuthill McKee [45] for graph mapping. It introduces the idea of graph mapping based on similarity metrics (e.g., bandwidth of the adjacency matrices) and offers an extensible framework to implement algorithms on InfiniBand, Ethernet, BlueGene, and Cray networks.

MPIPP [46] is a framework dedicated to MPI applications. Its goal it to reduce the communication costs between groups of processes. The original targets of this work are the meta-cluster architectures, but it could be adapted also in a context of multicore nodes if the node internal topology and organization information was to be gathered, which is currently not the case.

The *resource binding* technique is applied for MPI applications in several works. [47], as well as [48] use it to reduce communication costs in multicore nodes. Both works rely on the SCOTCH [22] partitioning software to compute the mapping. Also, Rodrigues et al. [47] use a purely quantitative approach while the approach in Mercier and Clet-Ortega [48] is qualitative since it uses the *structure* of the memory hierarchy of a node.

Brandfass et al. [7] also strive to reduce the communication costs of CFD MPI applications. It uses a so called *rank reordering* technique, but it is not the same technique that we described in a previous section of this chapter. Indeed, Brandfass et al. [7] reorganize the file containing the nodes' name (a.k.a. the hosts file), thus changing the way processes are dispatched on the nodes. By doing so, it manages to regroup physically processes that communicate a lot with each other. However, the processes are not bound to dedicated processing units and the application does not actually rely on the reordering mechanism available in the MPI standard (as shown in the next paragraph).

It is also possible to map several types of processing entities. This case occurs when so-called hybrid programming paradigm are used (e.g., message passing and multithreading). For instance, Dümmler et al. [49] explore the issue of hybrid, MPI + OpenMP application multithreaded process mapping.

In some cases, a thorough knowledge of the application is very helpful. For instance, Aktulga et al. [50] discuss works on topology aware mapping of an eigenvalue solver. They conducted an in-depth study of their application and have been able to propose a communication model based on the dilatation, the traffic and the congestion. They show that minimizing these factors by performing a relevant mapping induces execution time gains up to a factor of 2.5.

5.6.2.4 Topology Mapping Support in the Message Passing Interface As seen in the previous section, MPI applications are often the target of topology mapping techniques and frameworks. Actually, both leading free MPI-2 implementations, that is, MPICH2 [51] and Open MPI [52] provide options to bind the MPI processes at launch time thanks to their respective process manager (resp. Hydra and ORTE). The user can choose among some predefined placement policies (e.g., [53]). As for vendor implementations, several offer mechanisms that allow the use to better control

it execution environment by binding the MPI processes. Cray's [54, 55], HP's [56] and IBM's [57] MPI versions offer this possibility.

As a matter of fact, the MPI standard itself features several functions dealing with virtual topology building and management. Both Cartesian and generic process topologies can be created at the application level and several of the functions even possess a `reorder` parameter. However, as Träff [39] explains, the actual implementation of these routines are rather trivial and usually do nothing to reorder the processes, except in the case of a few vendor implementations, such as the one provided by NEC [39].

In revision 2.2 of the MPI standard, more scalable versions of the virtual topology creation and management routines have been introduced. For instance, it is the case of the `MPI_Dist_graph_create` function [44]. Implementations actually performing reordering of this routine are available: [8] relies on the Hwloc tool to gather hardware information and bind processes, while TREEMATCH is in charge of computing the reordering of MPI process ranks. LibTopoMap [5] also implements this part of the MPI interface with various strategies.

There are also specific parts of the MPI standard that can take advantage of an optimized process placement. I/O operations fall in this category, as the process placement can be performed in order to optimize the pattern of accesses to files between the processes. Venkatesan et al. [58] describe an implementation of such feature, and is based on an algorithm called *SetMatch*, which is a simplified version of the TREEMATCH algorithm.

5.6.2.5 Other Programming Models and Standards Other programming models also address the issue of virtual topology mapping. For instance, CHARM++ [59, 60] features optimizations for specific topologies [61]. CHARM++ also performs dynamic load-balancing of internal objects (chares). Such load balancing is done by the CHARM++ runtime system and does not require to modify specific parts of the application code. Moreover, the load balancer can be chosen and its parameters set at the beginning of the execution. Thanks to CHARM++ modularity, user-defined load-balancers can be easily added to the set of existing ones. There are several criteria to perform load-balancing. Among the possible ones, a topology-aware load-balancing, called NucoLB (non-uniform communication costs load balancer) has recently been proposed [62]. The idea is to gather the topology information and to dynamically monitor the volume of data exchanged by the chares. Then, the NucoLB load-balancer migrate the chares according to the topology and their affinity in order to reduce the communication cost of the application among and within compute nodes. Results show improvement up to 20% in execution time.

It is also possible to perform topology-aware mapping in PGAS languages (e.g., UPC [63]). PGAS languages expose a simple two-levels scheme (local and remote) for memory affinity that can be used to map the processes depending on the exchanged data and the underlying physical topology. Indeed, in some PGAS programs, it is possible to know the communication pattern based on the code and the distribution of the data arrays. With this information, it is natural to apply a process mapping algorithm to carefully map the processes onto the architecture.

5.7 Conclusion and Open Problems

In order for modern parallel machines to deliver their peak performance to applications, it is necessary to bridge the increasing gap between programming models and the underlying architecture. Among the various factors influencing the performance, process placement plays a key role as it impacts the communication time of the application. This problem has gained a huge momentum with the appearance of NUMA architectures as the communication cost between two processes can vary of several orders of magnitude depending on their location.

In this paper, we have surveyed different techniques, algorithms and tools to perform a topology-aware process placement. In all cases, the problem consist in matching the virtual topology (that may represents communication pattern of the application) to the physical topology of the architecture.

While there exist many solutions to address the topology mapping problem, we can list a set of open problems that shall need to be solved in order to reach larger scale machines.

- A first important issue is the ability to handle very large problems. Some high-performance computing applications feature hundred of thousands of processes. Mapping these processes onto the architecture require a huge computing power. It is therefore necessary to improve the scalability of the algorithms by reducing their complexity and implementing their most costly parts in parallel.

- Fault-tolerance is also an important issue as failures are becoming a "normal" feature of current parallel computers. Computing mappings that are able to cope with failures is therefore of high interest. A way to tackle this problem is to couple the fault-tolerant part and the mapping part of the runtime system in order to take joint decisions when necessary.

- Reducing the communication part has a huge impact on the energy consumption as between 25% and 50% of the energy spent is due to data movement. However, we are lacking studies about the real gain of topology-aware mapping and energy savings. Moreover, it should also be possible to devise energy-centric metrics for this specific problem.

- Many applications have a communication pattern that varies during the execution (dynamic). Its should be interesting to study how the mapping can be adapted, according to these changes. Several solutions could be tested from a global remapping requiring migration of the processes and changes of the internal organization of the application (e.g., MPI communicators) to local remapping within a NUMA node with the advantage of being able to distribute this remapping and doing it transparently, application wise.

- Extracting the communication pattern is a difficult task. It requires a thorough knowledge of the target application or to monitor it in order to extract its pattern. However, other techniques are possible such as source-code analysis through

compilation techniques or software engineering techniques (skeleton, component) that, by design, provides important information of the application behavior.

- Another important research issue is the link between the different aspects of process affinity: within node (cache), between nodes (network) and between node and storage. Each of these aspects may incur contradictory objectives in terms of placement. Therefore, it requires to find compromises or to be able to adapt, at runtime, the mapping according to the dominating factor.

Acknowledgments

This work is supported by the COST Action IC0805 "Open European Network for High Performance Computing on Complex Environments".

Bibliography

1. "Prace scientific case for hpc in europe 2012 – 2020," 2012. `http://www.prace-ri.eu/PRACE-The-Scientific-Case-for-HPC`.
2. J. Dongarra *et al.*, "The international exascale software project: A call to cooperative action by the global high-performance community," *International Journal of High Performance Computing Applications*, vol. 23, no. 4, pp. 309–322, 2009.
3. C. Ma, Y. M. Teo, V. March, N. Xiong, I. R. Pop, Y. X. He, and S. See, "An approach for matching communication patterns in parallel applications," in *Proceedings of 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS'09)*, (Rome, Italy), IEEE Computer Society Press, 2009.
4. B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell System Technical Journal*, vol. 49, no. 2, pp. 291–307, 1970.
5. T. Hoefler and M. Snir, "Generic topology mapping strategies for large-scale parallel architectures," in *ICS* (D. K. Lowenthal, B. R. de Supinski, and S. A. McKee, eds.), pp. 75–84, ACM, 2011.
6. F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "Hwloc: a generic framework for managing hardware affinities in HPC applications," in *Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP2010)*, (Pisa, Italia), IEEE Computer Society Press, 2010.
7. B. Brandfass, T. Alrutz, and T. Gerhold, "Rank reordering for MPI communication optimization," *Computer & Fluids*, vol. 80, pp. 372–380, 2013.
8. G. Mercier and E. Jeannot, "Improving MPI applications performance on multicore clusters with rank reordering," in *EuroMPI*, vol. 6960 of *Lecture Notes in Computer Science*, (Santorini, Greece), pp. 39–49, Springer, 2011.
9. T. Ogasawara, "NUMA-aware memory manager with dominant-thread-based copying GC," in *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, (New York, NY, USA), pp. 377–390, ACM, 2009.

10. Q. Yin and T. Roscoe, "VF2x: fast, efficient virtual network mapping for real testbed workloads," in *Testbeds and Research Infrastructure. Development of Networks and Communities* (T. Korakis, M. Zink, and M. Ott, eds.), vol. 44 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pp. 271–286, Springer Berlin Heidelberg, 2012.

11. T. Hatazaki, "Rank reordering strategy for MPI topology creation functions," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (V. Alexandrov and J. Dongarra, eds.), vol. 1497 of *Lecture Notes in Computer Science*, pp. 188–195, Springer Berlin / Heidelberg, 1998.

12. A. L. Rosenberg, "Issues in the study of graph embeddings," in *WG'80*, (London, UK), pp. 150–176, 1981.

13. S. Bokhari, "On the mapping problem," *IEEE Transactions on Computers*, vol. 30, no. 3, pp. 207–214, 1981.

14. S.-Y. Lee and J. K. Aggarwal, "A mapping strategy for parallel processing," *IEEE Transactions on Computers*, vol. 36, no. 4, pp. 433–442, 1987.

15. S. W. Bollinger and S. F. Midkiff, "Heuristic technique for processor and link assignment in multicomputers," *IEEE Transactions on Computers*, vol. 40, no. 3, pp. 325–333, 1991.

16. C. Sudheer and A. Srinivasan, "Optimization of the hop-byte metric for effective topology aware mapping," in *19th International Conference on High Performance Computing (HiPC)*, pp. 1–9, 2012.

17. H. D. Simon and S.-H. Teng, "How good is recursive bisection?," *SIAM Journal on Scientific Computing*, vol. 18, pp. 1436–1445, 1997.

18. G. Karypis and V. Kumar, "METIS – unstructured graph partitioning and sparse matrix ordering system, version 2.0," tech. rep., 1995.

19. K. Schloegel, G. Karypis, and V. Kumar, "Parallel multilevel algorithms for multi-constraint graph partitioning (distinguished paper)," in *Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, Euro-Par '00, (London, UK), pp. 296–310, Springer-Verlag, 2000.

20. B. Hendrickson and R. Leland, "The Chaco user's guide: Version 2.0," Tech. Rep. SAND94–2692, Sandia National Laboratory, 1994.

21. K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan, "Zoltan data management services for parallel dynamic applications," *Computing in Science and Engineering*, vol. 4, no. 2, pp. 90–97, 2002.

22. F. Pellegrini, "Static mapping by dual recursive bipartitioning of process and architecture graphs," in *Proceedings of SHPCC'94, Knoxville*, pp. 486–493, IEEE, 1994.

23. C. Walshaw and M. Cross, "JOSTLE: parallel multilevel graph-partitioning software – an overview," in *Mesh Partitioning Techniques and Domain Decomposition Techniques* (F. Magoules, ed.), pp. 27–58, Civil-Comp Ltd., 2007. (Invited chapter).

24. N. R. Adiga, M. A. Blumrich, D. Chen, P. Coteus, A. Gara, M. E. Giampapa, P. Heidelberger, S. Singh, B. D. Steinmacher-Burow, T. Takken, M. Tsao, and P. Vranas, "Blue Gene/L torus interconnection network," *IBM Journal of Research and Development*, vol. 49, no. 2, pp. 265–276, 2005.

25. D. Chen, N. A. Eisley, P. Heidelberger, R. M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. L. Satterfield, B. Steinmacher-Burow, and J. J. Parker, "The IBM Blue Gene/Q

interconnection network and message unit," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, (New York, NY, USA), pp. 26:1–26:10, ACM, 2011.

26. R. Alverson, D. Roweth, and L. Kaplan, "The Gemini system interconnect," in *Proceedings of the 2010 18th IEEE Symposium on High Performance Interconnects*, HOTI '10, (Washington, DC, USA), pp. 83–87, IEEE Computer Society, 2010.
27. Y. Ajima, S. Sumimoto, and T. Shimizu, "Tofu: A 6D mesh/torus interconnect for exascale computers," *IEEE Computer*, vol. 42, no. 11, pp. 36–40, 2009.
28. C. E. Leiserson, "Fat-trees: universal networks for hardware-efficient supercomputing," *IEEE Transactions on Computers*, vol. 34, no. 10, pp. 892–901, 1985.
29. F. Petrini and M. Vanneschi, "K-ary n-trees: High performance networks for massively parallel architectures," tech. rep., 1995.
30. S. R. Öhring, M. Ibel, S. K. Das, and M. J. Kumar, "On generalized fat trees," in *IPPS '95: Proceedings of the 9th International Symposium on Parallel Processing*, (Washington, DC, USA), p. 37, IEEE Computer Society, 1995.
31. J. Kim, W. Dally, S. Scott, and D. Abts, "Cost-efficient dragonfly topology for large-scale systems," *IEEE Micro*, vol. 29, no. 1, pp. 33–40, 2009.
32. B. Arimilli, R. Arimilli, V. Chung, S. Clark, W. Denzel, B. Drerup, T. Hoefler, J. Joyner, J. Lewis, J. Li, N. Ni, and R. Rajamony, "The PERCS high-performance interconnect," in *Proc. of 18th Symposium on High-Performance Interconnects (HotI'10)*, 2010.
33. G. Faanes, A. Bataineh, D. Roweth, T. Court, E. Froese, B. Alverson, T. Johnson, J. Kopnick, M. Higgins, and J. Reinhard, "Cray cascade: a scalable HPC system based on a Dragonfly network," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, (Los Alamitos, CA, USA), pp. 103:1–103:9, IEEE Computer Society Press, 2012.
34. P. Balaji, R. Gupta, A. Vishnu, and P. H. Beckman, "Mapping communication layouts to network hardware characteristics on massive-scale Blue Gene systems," *Computer Science – R&D*, vol. 26, no. 3-4, pp. 247–256, 2011.
35. B. E. Smith and B. Bode, "Performance effects of node mappings on the IBM BlueGene/L machine," in *Euro-Par* (J. C. Cunha and P. D. Medeiros, eds.), vol. 3648 of *Lecture Notes in Computer Science*, pp. 1005–1013, Springer, 2005.
36. H. Yu, I.-H. Chung, and J. E. Moreira, "Blue Gene system software – topology mapping for Blue Gene/L supercomputer," in *SC*, p. 116, ACM Press, 2006.
37. H. Subramoni, S. Potluri, K. Kandalla, B. Barth, J. Vienne, J. Keasler, K. Tomko, K. Schulz, A. Moody, and D. Panda, "Design of a scalable infiniband topology service to enable network-topology-aware placement of processes," in *Proceedings of the 2012 ACM/IEEE conference on Supercomputing (CDROM)*, (Salt Lake City, Utah, United States), p. 12, IEEE Computer Society, 2012.
38. M. J. Rashti, J. Green, P. Balaji, A. Afsahi, and W. Gropp, "Multi-core and network aware MPI topology functions," in *EuroMPI* (Y. Cotronis, A. Danalis, D. S. Nikolopoulos, and J. Dongarra, eds.), vol. 6960 of *Lecture Notes in Computer Science*, pp. 50–60, Springer, 2011.
39. J. L. Träff, "Implementing the MPI process topology mechanism," in *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, (Los Alamitos, CA, USA), pp. 1–14, IEEE Computer Society Press, 2002.

40. S. Ito, K. Goto, and K. Ono, "Automatically optimized core mapping to subdomains of domain decomposition method on multicore parallel environments," *Computer & Fluids*, vol. 80, pp. 88–93, 2013.

41. S. von Alfthan, I. Honkonen, and M. Palmroth, "Topology aware process mapping," in *Applied Parallel and Scientific Computing* (P. Manninen and P. Öster, eds.), vol. 7782 of *Lecture Notes in Computer Science*, pp. 297–308, Springer, 2013.

42. E. Jeannot and G. Mercier, "Near-optimal placement of MPI processes on hierarchical NUMA architectures," *Euro-Par 2010-Parallel Processing*, pp. 199–210, 2010.

43. E. Jeannot, G. Mercier, and F. Tessier, "Process placement in multicore clusters: Algorithmic issues and practical techniques," *IEEE Transactions on Parallel and Distributed Systems*, 2013. To be published.

44. T. Hoefler, R. Rabenseifner, H. Ritzdorf, B. R. de Supinski, R. Thakur, and J. L. Traeff, "The scalable process topology interface of MPI 2.2," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 4, pp. 293–310, 2010.

45. E. Cuthill and J. McKee, "Reducing the bandwidth of sparse symmetric matrices," in *Proceedings of the 1969 24th national conference*, ACM '69, (New York, NY, USA), pp. 157–172, ACM, 1969.

46. H. Chen, W. Chen, J. Huang, B. Robert, and H. Kuhn, "MPIPP: an automatic profile-guided parallel process placement toolset for SMP clusters and multiclusters," in *ICS* (G. K. Egan and Y. Muraoka, eds.), pp. 353–360, ACM, 2006.

47. E. Rodrigues, F. Madruga, P. Navaux, and J. Panetta, "Multicore aware process mapping and its impact on communication overhead of parallel applications," in *Proceedings of the IEEE Symp. on Comp. and Comm.*, pp. 811–817, 2009.

48. G. Mercier and J. Clet-Ortega, "Towards an efficient process placement policy for MPI applications in multicore environments," in *EuroPVM/MPI*, vol. 5759 of *Lecture Notes in Computer Science*, (Espoo, Finland), pp. 104–115, Springer, 2009.

49. J. Dümmler, T. Rauber, and G. Rünger, "Mapping algorithms for multiprocessor tasks on multi-core clusters," in *Proceedings of the 2008 37th International Conference on Parallel Processing*, pp. 141–148, 2008.

50. H. M. Aktulga, C. Yang, E. G. Ng, P. Maris, and J. P. Vary, "Topology-aware mappings for large-scale eigenvalue problems," in *Euro-Par 2012 Parallel Processing – 18th International Conference*, vol. 7484 of *Lecture Notes in Computer Science*, (Rhodes Island, Greece), pp. 830–842, 2012.

51. Argonne National Laboratory, "MPICH2." http://www.mcs.anl.gov/mpi/2004.

52. E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings of the 11th European PVM/MPI Users' Group Meeting*, (Budapest, Hungary), pp. 97–104, 2004.

53. J. Hursey, J. M. Squyres, and T. Dontje, "Locality-aware parallel process mapping for multi-core HPC systems," in *2011 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 527–531, IEEE, 2011.

54. National Institute for Computational Sciences, "MPI tips on Cray XT5." `http://www.nics.tennessee.edu/user-support/mpi-tips-for-cray-xt5`.

55. G. B. Justin L. Whitt and M. Fahey, "Cray MPT: MPI on the Cray XT," 2011. http://www.nccs.gov/wp-content/uploads/2011/03/MPT-OLCF11.pdf.

56. D. Solt, "A profile based approach for topology aware MPI rank placement," 2007. http://www.tlc2.uh.edu/hpcc07/Schedule/speakers/hpcc_hp-mpi_solt.ppt.

57. E. Duesterwald, R. W. Wisniewski, P. F. Sweeney, G. Cascaval, and S. E. Smith, "Method and system for optimizing communication in MPI programs for an execution environment," 2008. http://www.faqs.org/patents/app/20080288957.

58. V. Venkatesan, R. Anand, E. Gabriel, and J. Subhlok, "Optimized process placement for collective I/O operations," in *EuroMPI*, Lecture Notes in Computer Science, (Madrid, Spain), Springer, 2013. to appear.

59. L. Kale and S. Krishnan, "CHARM++: A portable concurrent object oriented system based on C++," in *Proceedings of Object-Oriented Programming, Systems, Languages and Applications (OOPSLA) 93*, pp. 91–108, ACM Press, 1993.

60. A. Bhatelé, L. V. Kalé, and S. Kumar, "Dynamic topology aware load balancing algorithms for molecular dynamics applications," in *ICS '09*, (New York, NY, USA), pp. 110–116, ACM, 2009.

61. A. Bhatel and L. V. Kal, "Benefits of topology aware mapping for mesh interconnects," *Parallel Processing Letters*, vol. 18, no. 04, pp. 549–566, 2008.

62. L. L. Pilla, C. P. Ribeiro, D. Cordeiro, C. Mei, A. Bhatele, P. O. Navaux, F. Broquedis, J.-F. Méhaut, and L. V. Kale, "A hierarchical approach for load balancing on parallel multi-core systems," in *41st International Conference on Parallel Processing (ICPP)*, pp. 118–127, IEEE, 2012.

63. U. Consortium, "UPC language specifications, v1.2," in *Lawrence Berkeley National Lab Tech Report LBNL-59208*, 2005.