Process mapping on any topology with TopoMatch

Emmanuel Jeannot

Process Mapping on any Topology with TopoMatch

Emmanuel Jeannot[1]

Talence, France

Abstract

Process mapping (or process placement) is a useful algorithmic technique to optimize the way applications are launched and executed onto a parallel machine. By taking into account the topology of the machine and the affinity between the processes, process mapping helps reducing the communication time of the whole parallel application. Here, we present TopoMatch, a generic and versatile library and algorithm to address the process placement problem. We describe its features and characteristics, and we report different use-cases that benefit from this tool. We also study the impact of different factors: sparsity of the input affinity matrix, trade-off between the speed and the quality of the mapping procedure as well as the impact of the uncertainty (noise) onto the input.

Keywords: Process Mapping ; Adaptative Algorithm ; Noise Analysis

1. Introduction

High-Performance Computing (HPC) is a solution to execute applications that have long sequential runtime or require a lot of memory. In HPC, applications are parallelized using different languages and runtime systems. To improve the efficiency of such applications, several types of optimizations can be conducted: at compile-time, at runtime or at launch time. In recent years, a lot of work has been done in the domain of compile time optimization (static scheduling chap. 5 of [1], vectorization [2], code transformation [3], etc.) as well as in the domain of runtime optimization (dynamic scheduling [1] chap. 15,

[1]Inria, Univ. Bordeaux, LaBRI

load-balancing [4], runtime systems [5], etc.). It appears that little research has been done in the domain of launch-time optimization that consists of optimizing the way the application is launched onto the different nodes of the target machine. Of course, job scheduling [6] is one obvious such research area and, in this work, we focus on the process placement (or process mapping) problem that is another topic for launch-time optimization.

The process placement problem consists of determining how the different processes (or even threads) of the application are allocated onto the different computing resources of the target machine by taking into account the *topology* of the machine and the *affinity* between the processes. The main idea is that, if two processes exchange a lot of data during the execution (and hence have a high affinity) they should be mapped to nodes that are close within the network or memory topology. The difficulty of this problem is that the affinity relationship is not *transitive*: process A can have a high affinity with process B, process B a high affinity with process C but this does not always imply that process A has high affinity with process C. Results and experiment from the literature have reported that huge gains in terms of performance can be achieved by a careful process placement at the launch of the execution (i.e. reduction of 80% of the congestion [7], up to 286% of execution time reduction for the LU NAS Benchmark in [8] or more than 25% of execution time reduction for a CFD application in [9]).

In this paper, we present a generic process mapping tool called TopoMatch. It is the offspring of an earlier tool called TreeMatch [9] that has been under development for more than 10 years. TopoMatch is a highly versatile tool as : (1) it can deal with any kind of topology (thanks to its internal use of the Scotch [10] graph partitioner); (2) it manages constraints (i.e. nodes that are not available, for the considered application, in the topology); (3) it deals with oversubscribing (i.e. mapping more than one process onto a compute unit); (4) it handles very large problem sizes (thanks to internal I/O optimization and fast mapping algorithm); and (5) it features adaptive algorithms able to explore different trade-off between speed and mapping quality. We have conducted

several experiments to assess the quality and speed of TopoMatch: the impact of the sparsity of the input affinity matrix and the study of the speed of the algorithm vs. the quality of the result trade-off. Moreover, several use-cases that use TopoMatch are described. Finally, a study of the noise sensibility of the input affinity matrix is conducted to show how the quality of such matrix is important to implement an efficient mapping.

This paper is organized as follows. In Section 2, we depict the related work. The background and problem definition are given in Sec. 3. TopoMatch is described in depth in Section 4. Experimental results are given in Section 5 before we conclude in Section 6

2. Related Work

Process placement is a topic that has drawn a lot of work in recent years (see [11] for a large overview.). One of the most cited work is LibTopoMap [7] that is a library on top of MPI to perform the mapping. It is thus less abstracted than a stand-alone tool that works for any kind of runtime system and programming model. In this regard, LibTopoMap does not support Numa-aware placement and it only deals with network topology (and not memory hierarchy). Generic approaches encompass MPIPP [12], that dispatches processes to compute resources. However, contrary to our approach, MPIPP is not topology-aware as it starts with a random initial guess and tries to improve it by swapping processes. An interesting approach, called geometric partitioning [13], uses the structure of the machine topology to compute a mapping of the processes. This approach is well suited for specific programs where the processes exhibits some structure (such as in a stencil case) while our approach is agnostic to the applications as long as we are able to gather the communication pattern.

HATS [14] targets heterogeneous environment and large-scale systems such as cloud with an approach using graph partitioning. Similarly, [15] looks at this problem from a quadratic assignment problem point-of-view. However, in these works, contrary to what we do here, no real execution experiments are

performed (only synthetic metric measurements).

EagerMap [8] is a mapping tool that exhibits a set of advanced algorithms for process mapping on hierarchical topologies. It shows performance similar or faster (for the NAS benchmarks) than TopoMatch for computing the mapping, however, contrary to what is presented here, it fails to handle large cases (1024 processes or more in our test), constraints and general topologies.

In general, a lot of existing works are restricted to specific topologies: : [16] is designed for grid and torus, [17, 18] for fat tree, [19] for mesh, [20] for Blue Gene machine, and TreeMatch [9] that deals with tree topologies and on which this work is partly based. The approach proposed in this paper, compared to these works, has the ability to tackle any kind of topologies.

3. Background

3.1. Problem Definition

The goal of the mapping problem is to allocate the processes (or threads) of a parallel application onto the available computing resources. In this paper, we do not distinguish between processes or threads as they can both be mapped on nodes, processors or cores. From an abstract point of view, the problem is the same. More precisely, it is formally defined as follows. It takes two inputs: a graph $G_T = (V_c \cup V_n, E_T)$ of the topology of a machine where some vertices (V_c) are compute nodes (or cores) and other vertexes (V_n) are network nodes (switches and routers). Theses nodes are connected through a network modeled by the set of edges (E_T). The other input is a valuated affinity graph $G_A = (V_p, E_A, C)$, where V_p is the set of processes/thread of the application and $C \colon E_A \to \mathbb{Q}$ is the affinity measure between two processes/threads (i.e. how each process of a pair must be mapped close to each other). The mapping problem consists of computing a function $\sigma \colon V_p \to V_c$ that allocates each process or thread of an application onto a compute node or core.

At a high level, the process mapping workflow requires several steps as described in Fig. 1: getting the topology (see sec. 3.2); getting the affinity (see 3.3);

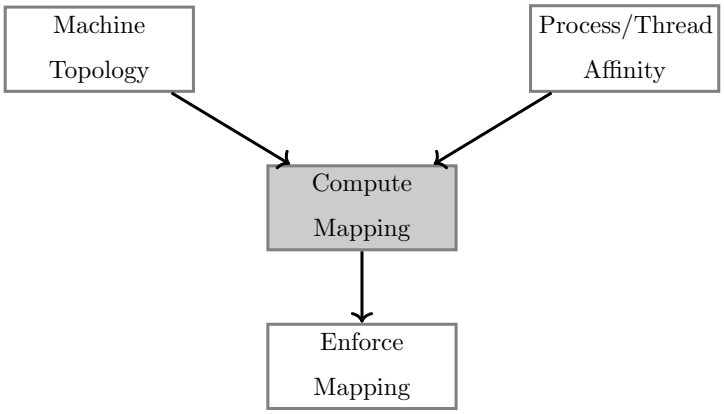


Figure 1: Complete Process Mapping Workflow

.

computing the mapping, which is the core of this article, and enforcing the mapping (see 3.4).

3.2. Getting the topology

Building the graph G_T requires to gather the topology of the system where the application is to run. For the node topology, a tool like Hwloc [21] can provide the description of the cache hierarchy and the core organization and numbering. For the network topology, Netloc [22] is able to gather a generic description of the topology in some cases (e.g. Clos network). Moreover, generic topologies can be represented in a compact way: this is the case of meshes, fat trees (such as Infiniband-based network), tori, etc. An example of fat tree represented in a compact way is given in Fig. 2. For general cases, it is required to have a way to describe any generic topology graph. In this work, we will use the decomposition-defined architecture format of the Scotch Library [10]. In general, building such a topology file requires the user to gather information about the machine where the application is to be executed[2].

[2]For instance from [23] we know that network of the Sunway TaihuLight supercomputer is composed of a 5-level hierarchy: 40 cabinets of 4 super nodes of 32 boards of 4 cards of 2 nodes for a total of 40 960 nodes

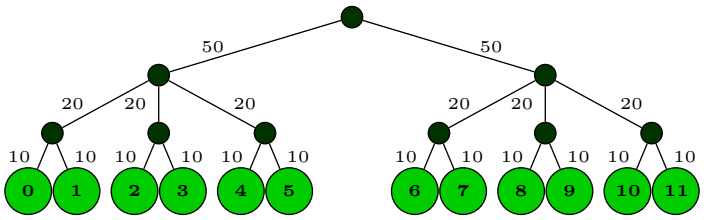


Figure 2: Fat tree topology. Only the leaf nodes (numbered from 0 to 11) are computational units (V_c set). Dark nodes model switches and routers (V_n set). Edge label model link bandwidth. Such a topology can be described in a concise manner by giving the arity of the 3 levels using for instance the Scotch tleaf format: `tleaf 3 2 50 3 20 2 10` – see Sec. 4.2.1

.

In general, the application cannot be mapped on all the nodes of the parallel machine. Only a subset of the nodes is available for the application (i.e. the nodes allocated by the resource manager). Hence, the application has only the freedom to map its processes or threads on these allocated resources. To handle this case, it is possible to restrict the topology to only the available nodes. However, in general this breaks the symmetry and regularity of the topology (for instance if an application is allowed to use only nodes 0, 1, 5 and 11 of Fig. 2, representing the resultant topology in a concise manner is difficult). A more elegant solution is to provide a set of constraints (e.g. a set of nodes) that, given a topology, restricts the mapping to these prescribed nodes. Handling constraints is a very important feature of a mapping tool and will be detailed in Section 4.4.

3.3. Measuring Affinity

The affinity graph encompasses the fact that some processes/threads should be mapped close to each other. Depending on the application and the programming model (process-based, thread-based, task-based etc.), it can cover different things: the volume of exchanged data, the number of messages – for process or task-based models – or the amount of shared memory pages – for thread-based model– , etc. The only assumption we make is that the higher the measure, the greater the affinity.

Computing the affinity between threads and processes is out of the scope of this paper. For the sake of completeness, we outline different methods: monitoring the application in cases where the affinity is the same from one run to another (this can be done by monitoring MPI communications [24]), statically computing the affinity based on the geometry of the computation (e.g. stencil code) or at runtime using information given by the pre-processing stage of the application. For instance, in the case of domain decomposition, each process is associated with a domain and the affinity between processes depends on the volume of data exchanged between each domain they work on).

3.4. Enforcing the mapping

TopoMatch is shipped with a mapping command-line executable that performs the "*Compute Mapping*" step and outputs a vector that tells for each process the ID of the processing unit where it has to be executed on the topology. TopoMatch is also a library that can be linked with any application and used at runtime to enforce the mapping as explained below.

Once the mapping has been computed, enforcing it can be done at runtime or at launch time. At runtime, this requires to change the thread mapping, using Hwloc or numactl. For process mapping, a better way to enforce it at launch time consists of performing rank reordering [25] as it does not require to migrate processes (more details are given in Section 4.6.2). At launch time, enforcing the mapping of MPI processes is easily done using the *rankfile* feature of the `mpirun` command line.

3.5. Metrics

In order to evaluate a placement, several metrics have been proposed. Let σ be the mapping function, i.e. $\sigma(i) \in V_c$ is the processing unit where process/thread $i \in V_p$ is placed. Let C be the affinity function, i.e. $C(i,j)$ is the affinity measure between processes i and j. Let d be the distance function on the topology graph i.e. $d(a,b)$ is the number of hops between nodes $a \in V_c$ and $b \in V_c$.

In the case where C represents a volume of data, the literature has proposed several mapping metrics that are the targets of the optimization (often minimization). Let B be the time to transfer one byte of data between two nodes: $B(\sigma(i), \sigma(j)) \times C(i, j)$ is the time for moving the data between process i and j after the mapping. We have three metrics:

- HopByte [20] is the accumulated cost of all the products between messages cost and the number of hops they have to traverse:

$$HB = \sum_{i,j} d(\sigma(i), \sigma(j))C(i, j)$$

- SumCom [26] is the sum of all the data movement cost:

$$SC = \sum_{i,j} B(\sigma(i), \sigma(j))C(i, j)$$

- MaxCom [27] is derived from SumCom and is the maximum of all the data movement cost:

$$MC = \max_{i,j} B(\sigma(i), \sigma(j))C(i, j)$$

In the case C represents something different than a volume of data (e.g. number of exchanged messages), we can still use the same formula (and keep their name), even if the semantic is different.

It has been shown that computing the mapping function σ to minimize any of these metrics is an NP-Hard problem [11]. In the following, we will focus on the Hop-Byte metric as it is agnostic to the way the distances are computed in the topology (no B function). We believe this to be a strong advantage, as gathering such information is error-prone, might be incomplete and is subject to inaccuracy. Moreover, in [27], we have shown that the HopByte seems to have a good correlation with performance and hence this metric is favored to evaluate the mapping.

4. TopoMatch Description

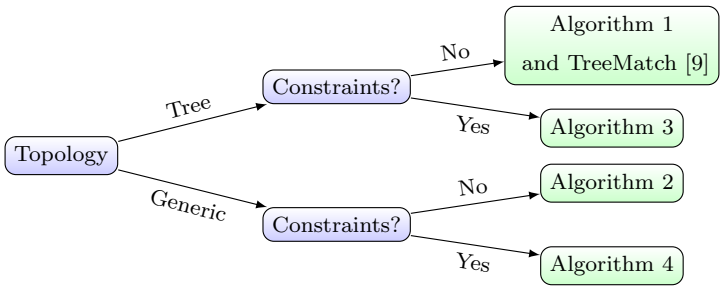


Figure 3: Default TopoMatch Algorithm Selection

.

4.1. Algorithms selection

TopoMatch features several algorithms depending on the type of topology and the necessity to deal with constraints. The default algorithm decision tree is described in Fig. 3 (only the top case is handled by TreeMatch and was published earlier in [9]). However, the user can enforce the use of generic topology algorithms in case of a tree topology (using an option of the command-line tool).

4.2. Managing Tree Topologies

4.2.1. Tree Topologies Format

A tree topology is composed of two types of nodes. The leaves of the tree are computing units and the other node of the tree defines the hierarchy of these computing units. An example of such topology is given in Fig. 2 where we have 12 compute nodes and a hierarchy of 3 levels. The hierarchy can represent the cache and memory hierarchy as well as the network (i.e. a fat tree) composed of a hierarchy of switches as it is the case in many of today's Infiniband based cluster. Moreover, the computing units modeled by the leaves can be of many types depending on the user needs (e.g. cores, sockets, nodes, etc.).

In TopoMatch, we use two kinds of format to represent a tree. The generic format is the Scotch `tleaf` whose syntax is as follows: `tleaf` $n\, a_1\, v_1 \ldots a_n\, v_n$ where $n+1$ is the number of levels of the hierarchy, a_i is the arity of level i and v_i is the valuation of the edges between level i and $i+1$. This syntax implies

that the tree is balanced and symmetric. This might not always be the case. In this case the constraint feature of TopoMatch described in Section 4.4 is designed, in particular, to address this issue.

The other format we use is the Hwloc format, given either in XML or by the Hwloc library (which is linked to TopoMatch). The Hwloc format is used to describe computing nodes and their cache hierarchy. Moreover, Hwloc can also generate synthetic topologies. In the end, both formats are equivalent except that using hwloc enables the management of physical and logical core numbering while tleaf format only allows for logical core numbering (which is sufficient in most cases).

4.2.2. Tree Topologies Algorithm

For tree topologies, we use an adaptive algorithm depending on the size of the input and the arity of the tree. TopoMatch is able to automatically switch form complex and very efficient algorithms (for small input) to very fast but less efficient algorithms for very large input. As it is a recursive strategy, the choice of the algorithm depends on the recursive step (simple at the beginning, more involved at the end).

To map the processes onto a tree topology we will build a hierarchy of groups of processes such that this hierarchy *matches* the tree topology. Processes in a group at the bottom of the hierarchy will have a high affinity and has the algorithm proceeds upward the affinity decreases. Once this hierarchy of processes is built the actual mapping is straightforward. We map the hierarchy of groups onto the tree topology from the top of the tree and when we arrive to the leaves the groups are actual processes and are naturally mapped to the corresponding computing unit.

To do so, the basic idea is to proceed from the bottom of the tree and to group processes according to their affinity. Hence, we have a new tree where leaves are groups of processes. We then proceed upward recursively by grouping such groups of processes. In other words, at the bottom of the tree processes are considered as groups of one element that are grouped and while we proceed

upward we make groups of groups according to their affinity. The algorithm is described in [9]. However, the grouping can be done in several ways. In order to adapt the grouping strategy to the difficulties (in terms of computational cost) that arise because of the number of possible groups and the arity we have devised the algorithm described in Alg. 1

When we are at the bottom of the tree, the grouping can be very costly as the number of candidate groups can be very large: it is $\binom{n}{a}$ where a is the size of the groups to be constructed (i.e. the arity of the considered level) and n is the number of processes (or groups computed at the previous iteration). The core of the algorithm is to select $m = n/a$ independent groups (groups that do not share processes) among all the candidate ones. For instance, from Fig. 2 at the first iteration we have $a = 2$, $n = 12$ and $m = 6$ (select 6 groups among 66) while at the second iteration $a = 3$, $n = 6$ and $m = 2$ (select 2 groups of group among 20). As we can see, the number of candidate groups decreases as we process the tree upward. Hence, in the case where the number of groups is larger than a threshold τ we have a set of fast strategies depending on the arity of the tree[3] and when this value is lower than τ we switch to slower but more efficient strategies.

In case the arity is two we have implemented a bucket grouping algorithm (line 4 of Alg. 1). Assume you have n processes and that your arity is 2. You have to find $m = n/2$ *independent*[4] groups of processes among $O(n^2)$ candidate groups. Each group of two processes is associated with the affinity value between the two processes of this group. The idea is then to sort these groups using this affinity in decreasing order. The goal is to ensure that groups with high affinity are likely to be selected first. In this case, the sorting dominates the complexity of this part as it is $O(n^2 \log(n^2))$. To speed up this part, we have

[3] By default, $\tau = 30\,000$ (as it empirically provides a good trade-off) but this value can be overwritten by the user using an option on the command-line tools or through via the library's API, see `https://gitlab.inria.fr/ejeannot/topomatch` for more details

[4] Two groups are independent if they do not share any processes

Number of processes	Partial sorting						Full sorting		
	Nb buck-ets	Sorted elements	Number of buckets used	Init time	Sort time	Grouping time	Init time	Sort time	Grouping time
4096	8	10251	1	0.14	0.004	0.16	0.11	7.55	7.68
8192	8	107234	1	0.56	0.04	0.62	0.45	26.08	36.60
16384	8	862567	1	2.41	0.48	2.99	5.32	1144.37	51.94
32768	16	22849	1	45.05	0.17	50.96	57.16	833.57	942.26

Table 1: Bucket grouping timing (in seconds). Partial sorting vs. full sorting comparison on an Intel Xeon CPU E5-2680 at 2.50GHz. The column "*sorted elements*" is the total number of the elements of the buckets used for the grouping

implemented a lazy randomized algorithm based on sampling and bucketing. We will build $k = 2^{\lceil \log_2(\log_2(n)) \rceil}$ buckets[5]. To do so, we take $r = 2^k$ random samples of the affinity matrix (hence $r = O(n)$). We sort these samples by decreasing affinity and then we assign $k + 1$ pivots p_i in order to be able to put all the values of the affinity matrix in the k buckets. Pivot values are as follows: $p_0 = +\infty$, $p_i = sample[2^i], 1 \leq i \leq k - 1$ and $p_k = 0$. Now, groups with affinity values between p_{i-1} and p_i are assigned to bucket $i \in [1, k]$. Then, we start from the first bucket (the one with the highest affinities), we sort it, and we consider each group in decreasing affinity[6]. Using this order, a new group is selected if none of its processes is in group that have already been selected: the group is independent of all the previously selected groups. The fact that, for computing the pivots, the samples, once sorted are not taken uniformly but using a geometric function has very strong advantages. The first bucket contains few but high values. Sorting such bucket is very fast and hence

[5]The number of buckets, k, is the largest power of two under $log_2(n)$

[6]The laziness of this algorithm means that we will process the remaining buckets only if we have not already selected the m groups

finding the independent groups within it is also very fast. Moreover, the last buckets contain a lot of small values. However, the process of selecting groups usually stops before considering these buckets (hence they are not sorted) as each process is put in a group early with the first buckets. This is what is highlighted in Table 1. In this table we see that in all the cases, we only had to build and sort the groups of the first bucket which is much faster than building and sorting all the groups (almost 19 times faster for the 32 768 processes case).

If the arity is lower than 5 we have implemented a fast grouping strategy (line 7 of Algorithm 1). It consists to built the m groups one after the other. To select one group, we greedily built 10 possible ones (excluding the groups that are not independent to the already selected ones) and choose the best one in terms of communication reduction.

The above fast grouping strategy is not efficient when we need to find a large number of groups. Hence, if the arity is strictly greater than 5 we use a k-way partitioning strategy (line 9 of Algorithm 1) that uses either Scotch[7] or a greedy partitioner that minimizes the cut.

When the total number of possible groups is lower than τ, we use a different method to compute the grouping (from line 11 to line 18 of Algorithm 1). Indeed, in this case the exploration space is small and we can use more evolved but more costly techniques. First, we compute a round robin mapping that consists of mapping process to the leftmost leaves of the tree using the identity ($\sigma(i) = i$). This gives an initial solution which is often the default solution of many runtime systems. Then, we try to improve this solution using three strategies that use the same pattern. We build all the possible groups, we sort these groups, and we select the first independent groups according to the order: two groups are independent if they do not share the same processes. As shown in [9], the graph of independent groups is a Kneser graph. Hence, the goal of the try_improve_sol procedure is to find a weighted maximum independent set of the

[7]We could have used other partitioners such as METIS [28]. However, Scotch as the required advantage to force balanced partition through the use of STRATBALANCE strategy

Algorithm 1: The Grouping Algorithm for Tree Topologies

```
Input: a// arity of the considered level of the tree
Input: n // number of processes (or groups) to map
Input: τ // Threshold (30 000 by default)
number_of_groups ← (n choose a) // # of candidate groups;
if number_of_groups > τ then
    if a = 2 then
        sol ← bucket_grouping();
    else
        if a ≤ 5 then
            sol ← fast_grouping();
        else
            sol ← k_partition_grouping();
else
    sol ← round_robin_mapping();
    groups ← enumerate_all_groups() // build all candidate groups;
    groups ← sort(groups, group_list_asc);
    sol ← try_improve_sol(sol, groups);
    groups ← sort(groups, group_list_dsc);
    sol ← try_improve_sol(sol, groups);
    groups ← sort(groups, weighted_degree);
    sol ← try_improve_sol(sol, groups);
return sol;
```

complement of such a graph: we look at the edges of this graph using the order given by the sort procedure. We then use the property of this particular kind the Kneser graph that states that every maximal independent set is maximum, meaning that, when we cannot increase the number of nodes in the independent set, we have a maximum set [9]. The three orders we use are: first groups with increasing amount of communication outside a group then, decreasing amount of communication. The last order is decreasing weighted degree: the average amount of communication outside of all groups that are not independent.

In conclusion, Algorithm 1 is an adaptive algorithm that allows performing an adaptive grouping depending on the difficulty of the problem. If the problem is very complex (i.e. the number of groups is very large), the algorithm computes a grouping very fast and as we go up in the tree, the problem becomes simpler and the algorithm becomes more evolved to compute a better grouping. This strategy has two advantages. First, it allows computing an overall solution very fast and second it performs the optimized grouping at the top of the tree which is the part that impacts the most the quality of the overall solution. The user has the possibility to change the value of τ. If τ is increased, faster but less precise algorithm will be used ; on the contrary if mapping time is less important than quality, it is possible to reduce the value of τ.

4.3. Managing Generic Topologies

In the case we do not deal with tree topologies, TopoMatch is still able to perform process placement. In this case it requires to use the Scotch library [10].

4.3.1. Generic Topologies Format

Linking Scotch with TopoMatch allows reading all the Scotch topologies. Basically, there are two kinds of topologies (called target) in Scotch.

The *Algorithmically-coded* architectures which represent, in a synthetic manner, most generic topologies that are present in standard HPC systems. We already described the tree leaf architecture in the Section 4.2.1. Similarly, Scotch uses its own format to synthetically describe torus (in any dimension), hyper-

cube, meshes (any dimension), etc. The *Decomposition-defined* architectures define topologies of any kind by explicitly describing the nodes and edges of the architecture.

4.3.2. Generic Topologies algorithm

Algorithm 2: The Scotch Partitioning Algorithm for Generic Topologies

Input: T `// The Scotch topology target`

Input: m `// The affinity matrix`

1 graph ← com_mat_to_scotch_graph(m,sparse_factor);

2 strat ← set_scotch_strategy(SCOTCH_STRATBALANCE);

3 partition ← SCOTCH_ComputeMapping (graph, T, strat);

4 return partition;

In the case TopoMatch uses the Scotch Library, the algorithm consists of transforming the affinity matrix into a Scotch graph and then setting the partitioning strategy to the "*balance*" one meaning that we want Scotch to evenly spread processes among the computing resources. Using the STRATBALACE strategy is necessary to ensure that each processing unit will receive exactly the requested amount of processes. If we would not use such strategy, it could happen that one processing unit receives two processes and another zero.

When transforming the affinity matrix into a Scotch graph (line 1 of Alg. 2), we need to build a representation of the matrix using the internal Scotch data structure. This representation is very efficient if the matrix is not dense. To improve Scotch performance, the user can set a *sparse_factor* (a value between 0 and 1) to only store, in this data structure, values strictly greater than $\mu\times$ sparse_factor (where μ is the maximum value of the matrix). The impact of the sparse factor on the quality and time of mapping will be studied in Sec. 5.1.

4.4. Handling Constraints

An important feature of TopoMatch is the possibility to compute the mapping on a subset of the architecture. In many cases, the available nodes for

Algorithm 3: The TopoMatchConst Algorithm for Generic Topologies

Input: T `// The topology tree`

Input: m `// The affinity matrix`

Input: C `// The constraints array`

$k \leftarrow$ arity at the top of the tree T.;

foreach i *in* $0..k-1$ **do**

  $n[i] \leftarrow$ number of nodes in subtree i that belongs to C

$p \leftarrow$k_partition(m, n, k) ; `// finds` k `partitions taking the constraints into account: partition` i `has` $n[i]$ `elements`

tab_m $\leftarrow$ split_com_mat(C, k, p); `// Splits the affinity matrix in` k `parts according to the partition just found above`

tab_C $\leftarrow$ split_constraints (C, k, T) ; `// Constructs a tab of constraints of size k: one for each partition`

if T *is not a leaf* **then**

  `// recursively calls` TopoMatchConst `on the` k `subtrees of the root of` T;

  foreach i *in* $0..k-1$ **do**

    TopoMatchConst (subtree i of T, tab_m[i], tab_C[i]).

  $r \leftarrow$ aggregates results of each subtrees;

else

  $r \leftarrow$ assigns the process/constraint to T;

returns r as result for T;

executing the application are only a subset of all the nodes of the parallel platform. To actually map processes to a subset of the available cores, TopoMatch implements the constraints feature. This feature consists of giving (through a file for the command line tool or through an array via the library's API, see `https://gitlab.inria.fr/ejeannot/topomatch` for more details) the list of nodes to which it is possible to map the processes. This list must be a subset of the IDs of the nodes of the whole topology. Internally, we have two cases and the algorithm works as follows. If the topology is a tree, the algorithm consists of partitioning from top to bottom the affinity matrix. Each partition is assigned to a given subtree of the topology. The size of the partitions is such that each subtree has exactly the required number of processes specified by the constraints. The partitioning is done such that the cut is minimized using either a greedy algorithm enriched with a balancing phase or the Scotch k-way partitioner if available. The algorithm is depicted in detail, in Alg. 3.

Algorithm 4: The TopoMatchConstScotch Algorithm for Generic Topologies and Constraints

Input: T `// The Scotch topology target`
Input: m `// The affinity matrix`
Input: C `// The constraints array`
1 SCOTCH_archInit(sub_arch);
2 SCOTCH_archSub(sub_arch T, $|C|$, C);
3 local_sol ← scotch_partitioning(sub_arch, m, $|C|$);
`// Renumber solution to change frame of reference;`
4 **foreach** i *in* $0..|C|-1$ **do**
5 global_sol[i] ← C[local_sol[i]];
6 return global_sol;

In the case of a general topology graph, we need to use the sub-arch feature of Scotch. The algorithm is depicted in Alg. 4. In this case, we build a sub architecture that corresponds to the nodes of the graphs given by the constraints. Then, we ask Scotch to compute the mapping of the affinity matrix

graph onto the sub architecture (using Algorithm 2). The solution is numbered in the frame of reference of the sub-architecture (i.e. from 0 to $|C| - 1$). We therefore renumber the solution to have it in the frame of reference of the global architecture.

4.5. Over- and under-subscribing

In many cases, the number of computing resources does not match the number of processes or threads to be mapped.

When the number of resources is larger than the number of processes, the internal algorithm selects the best resources to optimize the mapping and guarantee that no resource is allocated more than once.

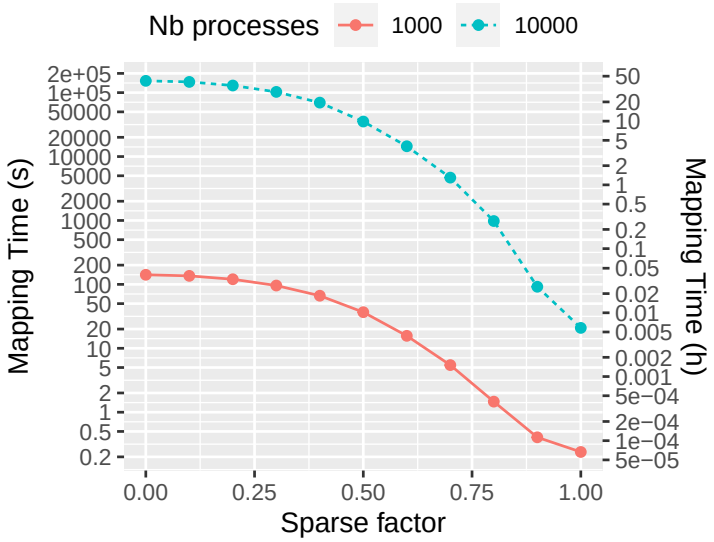
On the contrary in some cases, it is important to be able to map more than 1 process per computing unit. For instance, if a computing unit is a dual socket node, then mapping one process per socket (two per computing unit) is often a good strategy. In this case, TopoMatch features an *oversubscribing factor* f that allows mapping up to f processes per computing unit. To do so, we have two cases. First, if the input topology is a tree then TopoMatch adds a new layer of leaves to the tree (with f new leaves per leaf in the original topology) and call the standard algorithm. It then collapses the solution to match the original tree by carefully renumbering the mapping σ. If the input topology is not a tree or the Scotch partitioning is forced, the oversubscribing is ensured by enforcing the "*balanced*" mapping strategy and checking that the number of processes mapped to each resource does not exceed f.

4.6. Usage

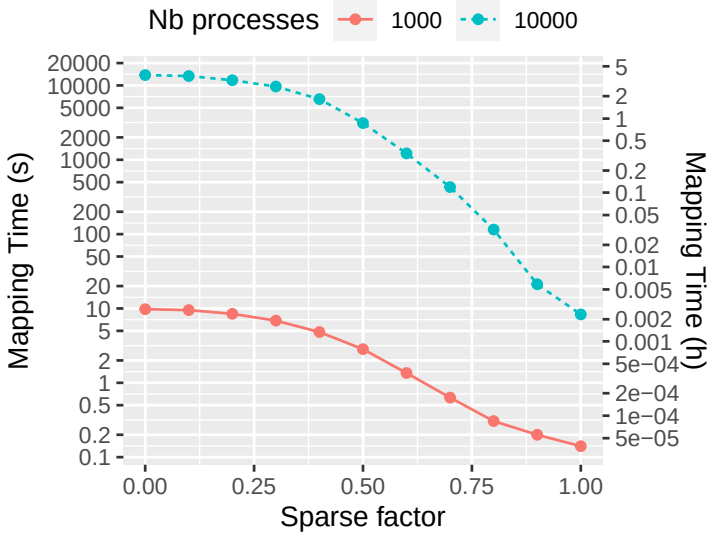
Tools for process mapping can be used in many contexts. In this section we outline several usages that have been designed and implemented with TopoMatch (or its earlier version TreeMatch).

4.6.1. Process Placement

The first obvious use case consists of mapping processes onto the topology in order to reduce the communication cost of the application [9, 11]. It requires to



(a) Fat tree (80640 PU)



(b) 3D mesh (50x50x50)

Figure 4: Scotch mapping time in function of the sparse factor for two topologies (left y-axis: second, right y-axis: hours))

get the communication pattern and the topology. Then, the mapping is enforced using tools provided by the application launcher. For instance, the OpenMPI

launcher (*mpirun*) implement the rankfile feature that allows defining where each MPI rank will be executed (on which node and which core).

4.6.2. Rank Reordering

A technique was depicted In [29, 25] that is useful in runtime systems where ranks are assigned to processes such as in MPI. In MPI, communicators are used to execute collective communications. In a communicator, each process has a rank that determines how it contributes to the communications. The rank reordering technique consists of changing the rank of each process at the beginning of the execution. This means that the role of the processes changes but not the mapping itself: rank reordering is the dual approach of process mapping. The strong advantage of this approach is that it does not require to migrate MPI processes. Therefore, it is useful to enforce the mapping at runtime. For instance, if we see that in an application rank i and rank j communicate a lot, it is better to reorder the ranks such that the processes of rank i and j are close in the topology [30]. This might require to exchange some data. However, such overhead pays-off if the application communication cost is sufficiently reduced.

4.6.3. Job Allocation

In parallel systems, the job scheduler has the role of allocating parallel applications to the different nodes of the machine. In many cases, the number of available nodes is larger than the number of requested nodes. In this case, an optimization consists of performing a topology-aware selection of the nodes taking into account the communication pattern of the application and performing the best choice among the available resources. This can be done using TopoMatch [31] thanks to its constraints feature. It works as follows. The job scheduler uses the communication pattern given by the user when the job is submitted, in addition to the standard pieces of information that it usually uses (number of nodes, duration, etc.). The job scheduler also knows the available nodes and the topology of the target platform. With all these inputs, TopoMatch computes a topology-aware process placement on the available nodes

using the constraint feature described in Sec. 4.4. The nodes are then allocated to the application accordingly.

4.6.4. Topology-Aware Load Balancing

Load balancing allows, at runtime, to optimize the execution of a parallel application by having each node executing roughly the same amount of work. Thanks to TopoMatch, load balancing can be further optimized as the elements that are balanced might communicate between them. This is the case, in the Charm++ model: "chares" are balanced between computing units, exchange data and communicate. In [32, 33] we have proposed a topology aware load balancer for Charm++ that uses TreeMatch/TopoMatch to take into account the topology and the affinity between the chares. Similar approaches have been proposed in [34, 35]

4.6.5. Topology-Aware Fault Tolerance

In [36] we performed elastic computations in case of node failures or when new nodes are available. The runtime system migrated MPI processes when the number of computing resources has changed. The TreeMatch/TopoMatch algorithm was used to recompute the process mapping onto the available resources taking into account the communication pattern gathered using an MPI introspection dynamic monitoring features described in [37]. The algorithm decides how to move processes, based on the communication matrix gathered by the application: the more two processes communicate, the closer they are remapped onto the physical resources.

5. Experimental Results

The experiments were carried out on two machines.

The first test machine is Plafrim 1, a 68 nodes machine with a fat-tree network. It is an InfiniBand QDR network made of four switches with 17 nodes each. Each node has two quad-core Intel Xeon X5550 processors

The second machine, Plafrim 2, is a 88 node machine with a fat-tree network. It is an InfiniBand QDR network made of four switches with 22 nodes each. Each node contains two Intel Xeon E5-2680 v3 processors (24 cores total, split in 4 NUMA nodes with 6 cores each). Hence, for the whole fat tree (including the cores), the arities, from the root to the leaves are 4, 22, 4 and 6.

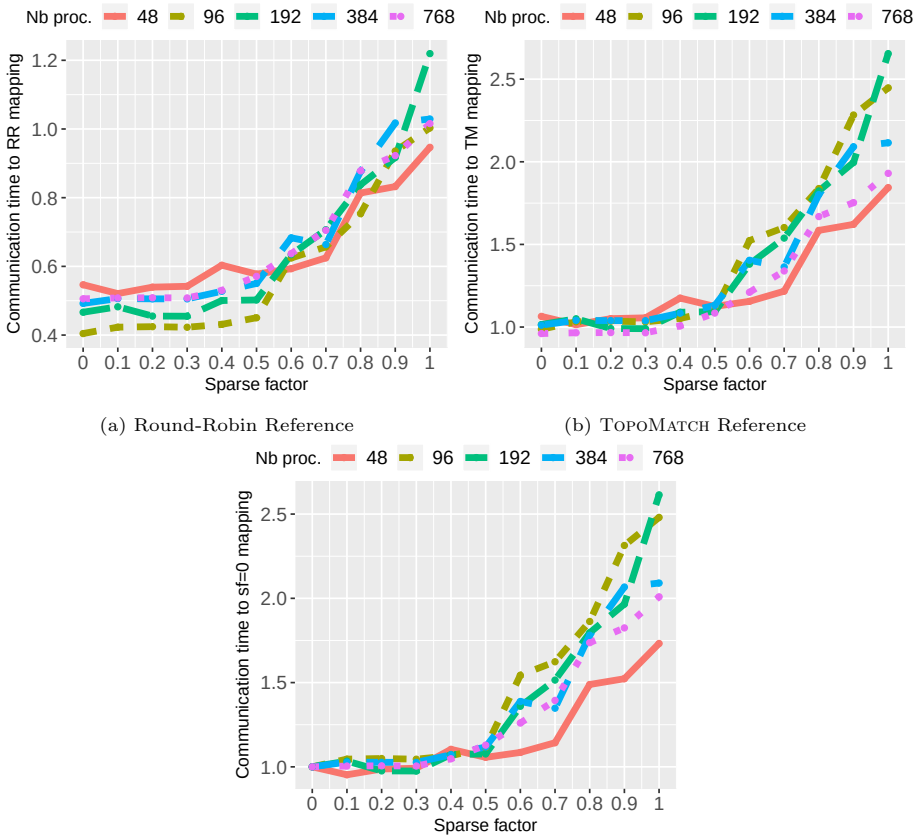
When we compare the mapping performance of TopoMatch versus the Round Robin mapping, we do not include the mapping computation time as this time is very short (less than 500 ms for 1024 processing units on an Intel Xeon Gold 6240 CPU at 2.60GHz) and it could be amortized after several executions.

When needed, the communication matrices are extracted from MPI application using the monitoring tool of OpenMPI [24] which does not require to modify the application. The advantage of this tool is that it provides the communication matrix once the collective communications have been decomposed in point-to-point yielding to a very precise pattern.

5.1. Impact of the sparse factor

In Fig. 4, we display the Scotch mapping time for two topologies (a fat tree of 86 400 leaves and a 3D mesh of 125 000 nodes) in function of the sparse factor: the higher the sparse factor the fewer entries are kept in the communication matrix (as explained in Sec. 4.3.2). We have taken two random (uniform trial) input matrices of order 1 000 and 10 000. We see that there is a more than 3 orders of magnitude between sparse factor of 0.9 (only the 10% largest values are kept) and a sparse factor of 0 (all values are considered). In comparison, the TopoMatch default strategy (for the fat tree case) has timings of around 0.97 second and 27.7 seconds for respectively 1 000 and 10 000 processes (which correspond to a sparse factor of around 0.9). Hence, being able to "*sparsify*" the input affinity matrix can help to gain several orders of magnitude to compute the mapping.

In Fig. 5, we study the impact of the sparse factor on the quality of the mapping. In this experiment, we emulate the communication cost of a given



(a) Round-Robin Reference

(b) TopoMatch Reference

(c) SF=0 reference

Figure 5: Communication Time Emulation Comparison of the Scotch Mapping with different sparse factors and Against Different References and for Various Number of MPI Processes on a Fat Tree Machine (Plafrim 2).

communication matrix using `MPI_Alltoallv` (each rank i sends $m(i,j)$ bytes to all ranks j) and do not perform any computation. Hence, we only measure communication time. The mapping is computed by Scotch (Alg. 2) with the prescribed sparse factor. We run the experiments using between 2 and 32 nodes (resp. 48 and 768 MPI processes) of the Plafrim 2 machine. We present the same result in Fig. 5a to Fig. 5c but with different references: In Fig 5a, we

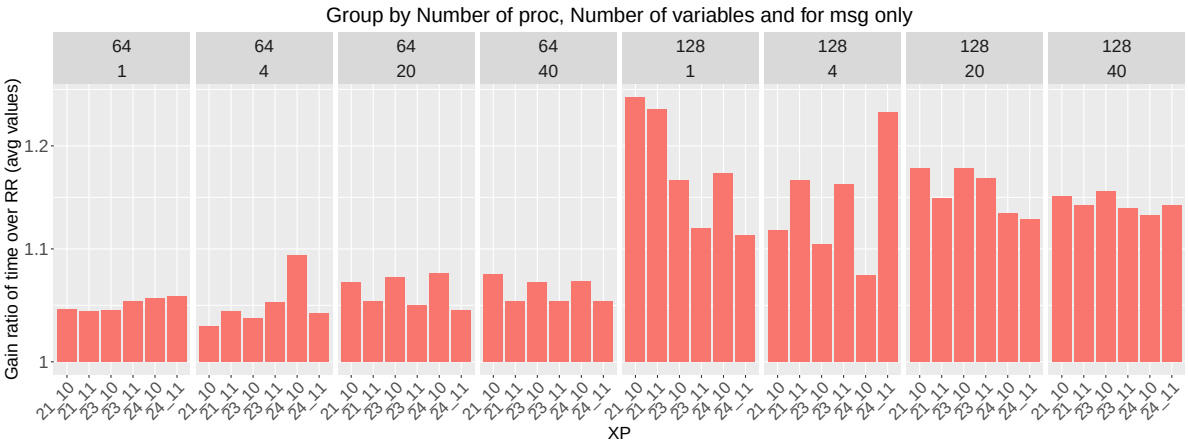


Figure 6: Gain over the Round Robin Placement of TopoMatch (using plain TreeMatch algorithm) for the MiniGhost Application. Bars are grouped by number of processes (64 or 128) and number of variables per points(from 1 to 40). Each individual bar is labeled , the type of stencil (21 = 2D5PT, 23= 3D7PT, 24 = 3D27PT) and the Boundary exchange implementation (10 = BSPMA, 11=SVAF). The size of the stencil ($n_x \times n_y \times n_z$) is 1000 for all cases. See [38] for more details. Absolute execution times vary between 8 and 53 seconds.

compare to the round robin mapping[8]. In Fig. 5b, we compare to the default TopoMatch mapping (using the original TreeMatch algorithm without Scotch) and in Fig. 5c, we compare to the *sparse factor = 0* case: how much does the communication is improved compared to the case where SF = 0 (using Alg. 2 with the complete affinity matrix). The measured communication time is the maximum of all the `MPI_Alltoallv` involved and we display an average of 10 measurements. For a given sparse factor, a ratio r means that the benchmark has taken r more time with TopoMatch than with the reference.

Results show that the sparse factor below 0.5 does not impact the quality of the mapping. The quality then degrades and becomes worse than the Round-Robin (RR) mapping (Fig. 5a) for sparse factor larger than 0.9. In Fig. 5b, we see that a sparse factor between 0 and 0.5 leads to similar performance (ratio

[8]The *round robin* mapping consists of mapping process i to computing resource i: it is the default binding policy of MPI

between 0.96 and 1.17) compared to the TopoMatch mapping. Also, in Fig. 5c we do not see a trend in the number of MPI processes and in many cases the degradation with a sparse factor of 0.9 can be twofold or more. Moreover, the mapping time is greatly reduced when we go from a sparse factor of 0 to a sparse factor of 0.5 (4x compute time reduction). For these reasons, the default sparse factor value is set to 0.5 in TopoMatch (but it can be changed by the user).

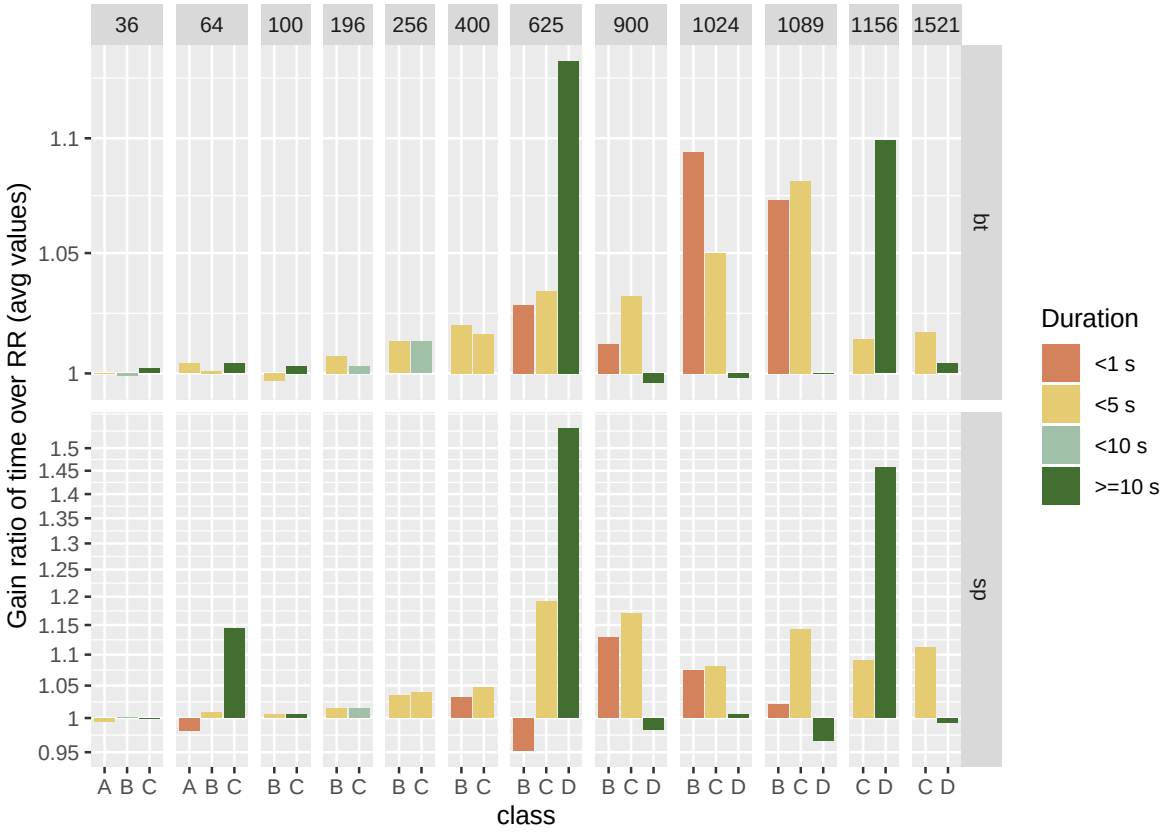


Figure 7: Gain over the Round Robin Placement of TopoMatch (using plain TreeMatch algorithm) for the SP and BT kernels of the NAS parallel benchmarks and different number of processors (square number of processors) and different class (A to D). Colors of the bars are related to the duration of the Round Robin Case.

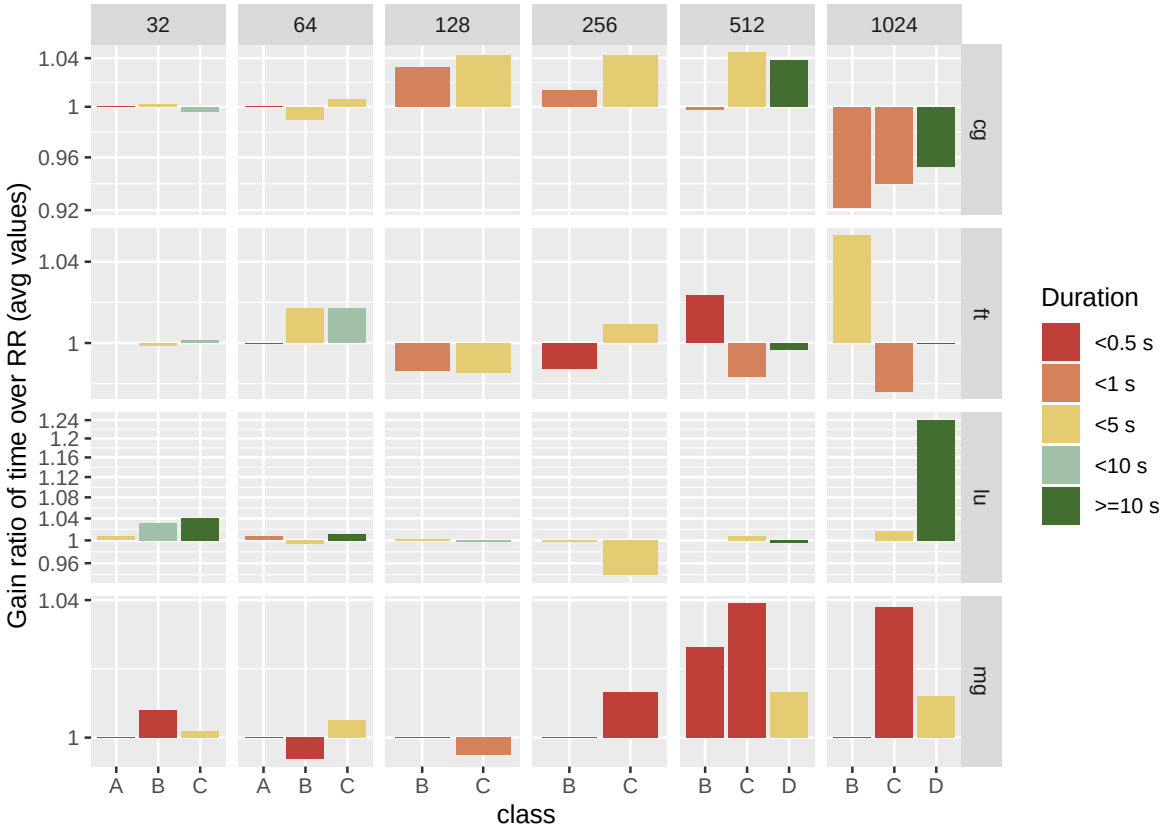


Figure 8: Gain over the Round Robin Placement of TopoMatch (using plain TreeMatch algorithm) for the CG, FT, LU and MG kernels of the NAS parallel benchmarks and different number of processors (power of two number of processors) and different classes (A to D). Colors of the bars are related to the duration of the Round Robin Case

5.2. Mapping Stencil Application

In this section, we report experiments carried out with a stencil mini-application called MiniGhost [38]. Its main advantage is the possibility to set many parameters that have an influence on the ratio of computation vs. communication (number of variables by stencil points, number of iterations, size of the problem, number of cells by process, etc.) and for which the structure of the execution can be tuned to see different behavior of mapping strategy (stencil dimensions 2D or 3D, stencil size in each dimension, connectivity between stencil point, com-

munication strategy). We have also chosen MiniGhost as there is no overlap between communication and computation.

In Fig. 6, we plot the gain of TopoMatch against the Round Robin placement on the Plafrim 1 machine using between 8 and 16 nodes (64 and 128 MPI processes). Different number of variables are used on each point of the stencil as well as different stencil type and communication strategies. We see that the TopoMatch mapping is able to achieve gain up to 45% improvement with an average of 9% compared to the standard RR mapping.

5.3. Mapping NAS Parallel Benchmarks

In Fig 7 we depict the gain obtained by TopoMatch on the SP and BT kernels of the NAS Parallel benchmarks. We have used from 36 to 1521 processes (square number of processes [39]) of the Plafrim 2 machine and different classes (A to D). As classes are related to the problem size, we use A class for a number of processes lower than 100 and class D for a number of processes greater than 625. The color of the bars are related to the duration of the Round Robin placement. The ratio is computed using 10 runs. The ratio is 1, when bars are not visible. We see that for these kernels using TopoMatch enable consistent gains up to 15%. We also observe some performance losses that are due to the fact that the communication matrix does not capture all the phases of these kernels. However these losses are limited to 5% in the worst case.

In Fig. 8, we plot the same experiments as above but for different kernels[9] (the ones that require a power of 2 number of processes). Here, the results are less spectacular. The FT and MG kernels barely exhibit gains. For CG on 1024 processes, we see a slight degradation of the performance (limited to 5%). Only the case of the LU kernel (class D on 1024 processes) exhibits a significant gain of 25%.

In conclusion, we can see that not all applications are sensitive to process

[9] Due to internal limits on the way the LU kernel is written, its class B cannot be executed on more than 256 processes

placement. This depends on the way it is coded (some parallel program can have , by construction, a good locality), and on the way the communication matrix captures the actual affinity between processes. This means that the user

Topology	kernel	class	# Proc	SumComm TM	SumComm TM+const	SumCom RR	Ratio	Ratio Const
2D mesh 8x8	cg	C	64	1.55322e+11	1.61843e+11	2.32392e+11	0.67	0.70
2D mesh 20x20	cg	C	256	1.50752e+12	1.25024e+12	2.24214e+12	0.67	0.56
10D hypercude	cg	C	64	8.53695e+10	9.24831e+10	1.13824e+11	0.75	0.81
10 Dhypercude	cg	C	256	5.37686e+11	5.62584e+11	6.07044e+11	0.89	0.93
3D torus 2x4x8	cg	C	64	1.03746e+11	1.12046e+11	1.4228e+11	0.73	0.79
3D torus 8x4x8	cg	C	128	1.51185e+11	1.67784e+11	2.32411e+11	0.65	0.72
3D torus 8x4x8	cg	C	256	6.41426e+11	7.30942e+11	9.86436e+11	0.65	0.74
3D mesh 8x4x8	cg	C	128	1.99799e+11	1.74897e+11	2.46638e+11	0.81	0.71
3D mesh 8x8x8	cg	C	256	9.54426e+11	8.2638e+11	1.10025e+12	0.87	0.75
3D mesh 11x11x11	cg	C	1024	8.09822e+11	7.43324e+11	1.37142e+12	0.59	0.54
3D mesh 50x50x50	cg	C	1024	3.45528e+12	1.3052e+12	2.69805e+12	1.28	0.48
2D mesh 8x8	lu	C	64	2.21288e+10	2.86276e+10	2.21288e+10	1.00	1.29
2D mesh 20x20	lu	C	256	2.08085e+11	1.97101e+11	4.22708e+11	0.49	0.47
hypercude 10	lu	C	64	2.21287e+10	2.21287e+10	3.47735e+10	0.64	0.64
hypercude 10	lu	C	256	9.4837e+10	1.05967e+11	1.64384e+11	0.58	0.64
3D torus 2x4x8	lu	C	64	2.36962e+10	2.92548e+10	2.68705e+10	0.88	1.09
3D torus 8x4x8	lu	C	128	4.896e+10	5.76013e+10	5.21605e+10	0.94	1.10
3D torus 8x4x8	lu	C	256	1.2159e+11	1.36717e+11	1.67545e+11	0.73	0.82
3D mesh 8x4x8	lu	C	128	5.62032e+10	6.42369e+10	6.16442e+10	0.91	1.04
3D mesh 8x8x8	lu	C	256	1.51582e+11	1.37384e+11	2.11802e+11	0.72	0.65
3D mesh 11x11x11	lu	C	1024	9.83983e+10	9.72757e+10	2.03649e+11	0.48	0.48
3D mesh 50x50x50	lu	C	1024	3.97028e+11	1.38743e+11	6.2864e+11	0.63	0.22

Table 2: SumCom of the Mapping for Different Topologies and Different Communication Matrix. We compare to the Round-Robin Case with and without Constraints

has to check whether or not, its application is suited for an optimized mapping. But in some cases substantial performance gains (up to 25%) are exhibited.

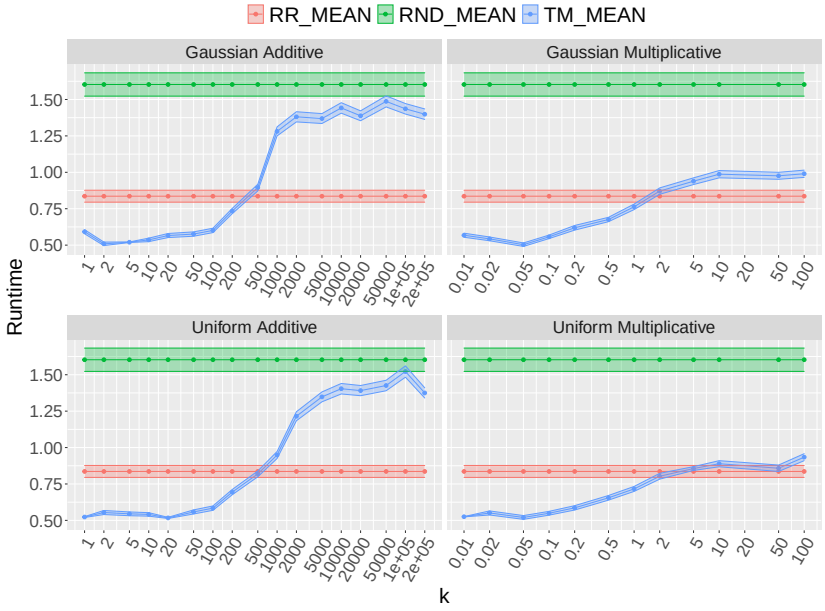
5.4. Computing the Mapping on Arbitrary Topologies

In table 2 we display the SumCom metric of the mapping computed by TopoMatch (with and without constraints) vs. the SumCom metric for the Round Robin placement for different topologies and different number of processes and different communication matrices. For the constraint case, we force TopoMatch to use the first n processing units (as for the round-robin case). By definition, this metric has to be lowered. We see that TopoMatch is able to handle various topologies and that the mapping it computes show a smaller SumCom than the Round Robin one (the only exception is when using a very large topology –e.g. 3D mesh of 50x50x50– where Scotch struggles in managing the combinatorics). We compare the case with and without constraints in order to compare the approach proposed by Algorithm 2 and Algorithm 4 of Fig 3. Here, we see that in almost all the cases, TopoMatch behaves as expected: constraining the mapping to the same processing units as the round-robin case degrades the performance of the plain TopoMatch algorithm (i.e. without any constraints) but still provide a better mapping than the round-robin case (TopoMatch is able to find a better mapping than the round-robin one on the same set of processing elements). In some cases, TopoMatch is able to decrease the SumCom metric by more than 50% showing the versatility and efficiency of TopoMatch.

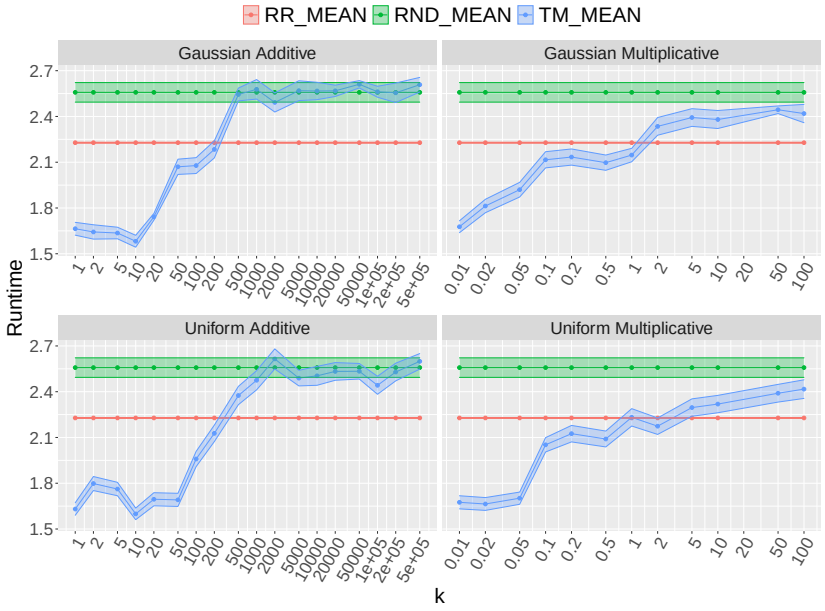
5.5. Noise Sensibility

An interesting question lies in the fact that, in many cases, the affinity matrix is not known perfectly due to an error of measurements or the impossibility of getting all the communications that occur within the applications.

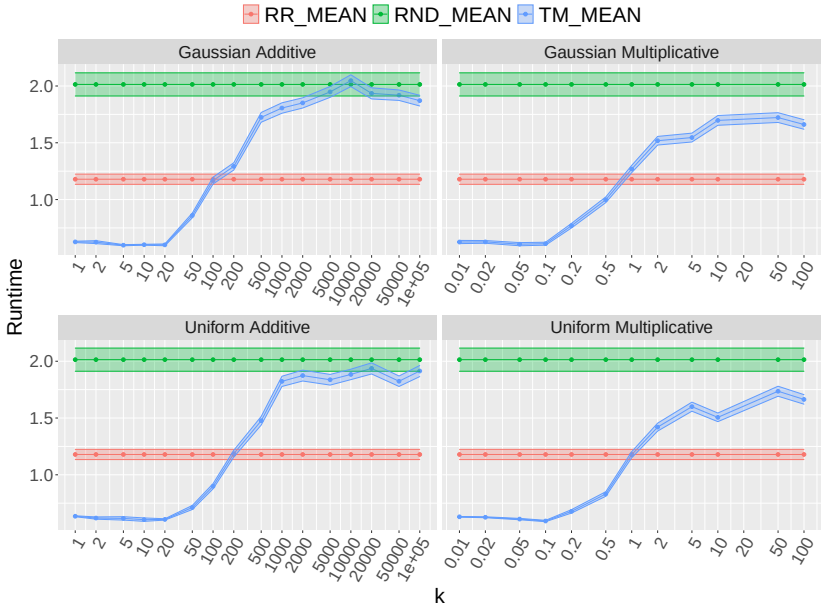
We have tested several models of noise. If m is the affinity matrix then each element (i, j) is updated to form a noisy affinity matrix $\tilde{m}$. We have two kinds of noise (additive noise and multiplicative noise) and two models (normal and uniform) this leads to:



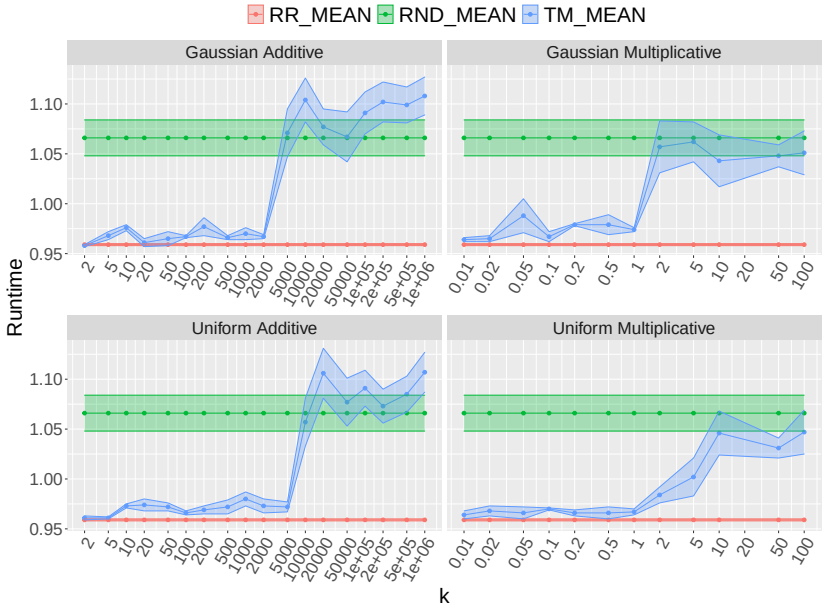
(a) 10 40 120 4 96 23 10



(b) 40 400 10 1 96 21 10



(c) 5 120 120 4 96 21 10



(d) 50 4 40 40 48 21 10 11

Figure 9: Impact of the Noise (the k parameter set the noise intensity) of the Input Affinity Matrix on the Performance of TopoMatch for the MiniGhost application. The different cases are presented by number of variables, size of the stencil (nx, ny, nz), number of processes, type of stencil and type of communication (see [38] for more details).

1. $\tilde{m} \leftarrow m + \mathcal{N}(0, k)$ (Gaussian Additive),

2. $\tilde{m} \leftarrow m + m * \mathcal{N}(0, k)$ (Gaussian Multiplicative),

3. $\tilde{m} \leftarrow m + \mathcal{U}(-k, k)$ (Uniform Additive).

4. $\tilde{m} \leftarrow m + m * \mathcal{U}(-k, k)$ (Uniform Multiplicative),

Where k is the parameter setting the noise intensity, $\mathcal{N}(0, k)$ is a random variable that follows a normal distribution of mean 0 and standard deviation of k and $\mathcal{U}(-k, k)$ is a random variable that follows a continuous uniform distribution in $[-k, k]$. Once the noise has been added, negative entries in the noisy affinity matrix are truncated to 0.

In Fig. 9 we show the result of experiments on the Plafrim 2 machine and the MiniGhost application with 48 or 96 MPI processes (2 or 4 nodes) and the 4 kinds of noise for representative MiniGhost settings. The x-axis represents the k parameter described above. On each part of the figure, we plot 3 curves. The 'RND-MEAN' curve corresponds to the average runtime of random mappings. The 'RR-MEAN' corresponds to the standard round-robin mapping. These two mappings do not depend on the value of k as they do not use the affinity matrix. The 'TM-MEAN' mapping corresponds to the mapping computed by TopoMatch once we have applied the noise using the value k which is represented on the x-axis. Hence, when k is 0, we have the standard TopoMatch mapping and the greater k the more noise is added to the input affinity matrix. We run several experiments for each value of k. The number of experiments is such that the precision of the 95% confidence interval (computed with the Student t-test) is less than 10% (i.e. the 95% confidence interval width divided by the mean is less than 0.1). The 95% confidence interval is displayed as a shaded ribbon. The trend of the curves behaves as expected. When k is small the mapping is efficient and as k increases, the performance decreases up to the point it reaches performance close to a complete random mapping.

More precisely, we see that for the case of multiplicative noise we obtain similar results whatever the MiniGhost setting. These correspond to model of

noise where only non-zero values are modified. We see that the normal case is more sensitive to the small value of k. This is due to the fact that for $\mathcal{N}(0, k)$ 68.3% of the values are in $[-k, k]$ and 95.4% are in $[-2k, 2k]$. This means that a significant portion of the values are increased by a larger factor in the case of the normal distribution than for the uniform distribution. However, when the noise is very large many values are truncated to zero and, in this case, the affinity matrix has too little information to guide TopoMatch. If TopoMatch does not have relevant input from the affinity matrix, it falls back to the round-robin strategy (see line 11 of Algorithm 1). This is what we see in the case of multiplicative noise: the performance degradation rarely reaches the full random case. On the contrary, for the additive case all the values are affected and the performance degrades only for large values of k : the impact is seen only when k is close to the largest values of the input matrix. Moreover, when k is very large, the matrix is highly random and the performances are similar to random placements.

We also see that in the case of 48 processes (Fig. 9d), there is no difference between the round-robin case and the TopoMatch case. This is due to the fact that for 48 processes, we only use two nodes and the difference between round-robin and TopoMatch is not visible as most communications are done within a node.

6. Conclusion

Process mapping is an important algorithmic problem that enables the optimization of the way an application is launched and executed. In this paper we have presented TopoMatch that is a tool, we have developed for more than 10 years since we initiated the TreeMatch project. TopoMatch is a versatile and generic tool to address the process mapping problem on any kind of topologies. It features an adaptive set of algorithms to handle very large problems and to provide a trade-off between the quality and the speed of the mapping. It manages constraints and handles over- or under-subscribing of the resources.

Several use-cases have been presented along with several experiments to study the impact of the "*sparsification*" as well as the sensibility of the noise in the input target matrix. Results show that keeping values greater than half of the largest value is sufficient to get good performances and, as soon as the noise is not on the same order of magnitude as the largest data of the input matrix, the mapping strategy provides similar results as the bottom-line case.

TopoMatch and its documentation is available as an open source software here: `https://gitlab.inria.fr/ejeannot/topomatch`

Acknowledgment

We would like to thanks François Tessier, Laurent Dutertre, Guillaume Mercier, Pierre Célor, Thibaut Lausecker, Adèle Villiermet, Farouk Mansouri, Fatima-Zahra El Akkary for their contribution to TreeMatch. We would also like to thank François Pellegrini for useful discussion about the Scotch Library and Clément Foyer for his feedback about this article.

References

[1] J. Blazewicz, K. Ecker, E. Pesch, G. Schmidt, J. Weglarz, Handbook on scheduling, Springer, 2019.

[2] M. Weinhardt, W. Luk, Pipeline vectorization, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 20 (2) (2001) 234–248.

[3] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, O. Temam, Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies, International Journal of Parallel Programming 34 (3) (2006) 261–317.

[4] G. Cybenko, Dynamic load balancing for distributed memory multiprocessors, Journal of parallel and distributed computing 7 (2) (1989) 279–301.

[5] C. Augonnet, S. Thibault, R. Namyst, P.-A. Wacrenier, Starpu: a unified platform for task scheduling on heterogeneous multicore architectures, Concurrency and Computation: Practice and Experience 23 (2) (2011) 187–198.

[6] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, Parallel job scheduling—a status report, in: Workshop on Job Scheduling Strategies for Parallel Processing, Springer, 2004, pp. 1–16.

[7] T. Hoefler, M. Snir, Generic Topology Mapping Strategies for Large-scale Parallel Architectures, in: Proceedings of the 2011 ACM International Conference on Supercomputing (ICS'11), ACM, 2011, pp. 75–85.

[8] E. H. Cruz, M. Diener, L. L. Pilla, P. O. Navaux, Eagermap: a task mapping algorithm to improve communication and load balancing in clusters of multicore systems, ACM Transactions on Parallel Computing (TOPC) 5 (4) (2019) 1–24.

[9] E. Jeannot, G. Mercier, F. Tessier, Process Placement in Multicore Clusters: Algorithmic Issues and Practical Techniques, IEEE Transactions on Parallel and Distributed Systems 25 (4) (2014) 993–1002. doi:10.1109/TPDS.2013.104.

[10] F. Pellegrini, J. Roman, Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs, in: High-Performance Computing and Networking, Springer, 1996, pp. 493–498.

[11] T. Hoefler, E. Jeannot, G. Mercier, An overview of topology mapping algorithms and techniques in high-performance computing, High-Performance Computing on Complex Environments 95 (2014) 75.

[12] H. Chen, W. Chen, J. Huang, B. Robert, H. Kuhn, Mpipp: an automatic profile-guided parallel process placement toolset for smp clusters and multiclusters, in: Proceedings of the 20th annual international conference on Supercomputing, 2006, pp. 353–360.

[13] M. Deveci, S. Rajamanickam, V. J. Leung, K. Pedretti, S. L. Olivier, D. P. Bunde, U. V. Çatalyürek, K. Devine, Exploiting geometric partitioning in task mapping for parallel computers, in: 2014 IEEE 28th International Parallel and Distributed Processing Symposium, 2014, pp. 27–36.

[14] S. Z. Koohi, N. A. W. A. Hamid, M. Othman, G. Ibragimov, Hats: Heterogeneity-aware task scheduling, IEEE Transactions on Cloud Computing (2022) 1–12doi:10.1109/TCC.2022.3184081.

[15] K. V. Kirchbach, C. Schulz, J. L. Träff, Better process mapping and sparse quadratic assignment, Journal of Experimental Algorithmics (JEA) 25 (2020) 1–19.

[16] R. Glantz, H. Meyerhenke, A. Noe, Algorithms for mapping parallel processes onto grid and torus architectures, in: 2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, IEEE, 2015, pp. 236–243.

[17] H. Subramoni, S. Potluri, K. Kandalla, B. Barth, J. Vienne, J. Keasler, K. Tomko, K. Schulz, A. Moody, D. K. Panda, Design of a scalable infiniband topology service to enable network-topology-aware placement of processes, in: SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, IEEE, 2012, pp. 1–12.

[18] G. Michelogiannakis, K. Z. Ibrahim, J. Shalf, J. J. Wilke, S. Knight, J. P. Kenny, Aphid: Hierarchical task placement to enable a tapered fat tree topology for lower power and cost in hpc networks, in: 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), 2017, pp. 228–237. doi:10.1109/CCGRID.2017.33.

[19] A. Bhatele, L. V. Kale, Heuristic-based techniques for mapping irregular communication graphs to mesh topologies, in: 2011 IEEE International Conference on High Performance Computing and Communications, IEEE, 2011, pp. 765–771.

[20] H. Yu, I.-H. Chung, J. Moreira, Topology mapping for blue gene/l supercomputer, in: Proceedings of the 2006 ACM/IEEE conference on Supercomputing, 2006, pp. 116–es.

[21] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, R. Namyst, hwloc: A generic framework for managing hardware affinities in hpc applications, in: Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on, IEEE, 2010, pp. 180–186.

[22] B. Goglin, J. Hursey, J. M. Squyres, Netloc: Towards a comprehensive view of the hpc system topology, in: 2014 43rd International Conference on Parallel Processing Workshops, IEEE, 2014, pp. 216–225.

[23] J. Dongarra, Report on the sunway taihulight system, PDF). www. netlib. org. Retrieved June 20.

[24] G. Bosilca, C. Foyer, E. Jeannot, G. Mercier, G. Papauré, Online Dynamic Monitoring of MPI Communications, in: Springer (Ed.), Euro-Par 2017: Parallel Processing - 23rd International Conference on Parallel and Distributed Computing, Vol. 10417 of LNCS, Santiago de Compostela, Spain, 2017, pp. 49–62.

[25] G. Mercier, E. Jeannot, Improving mpi applications performance on multicore clusters with rank reordering, in: Proceedings of the 16th International EuroMPI Conference, LNCS 6960, Springer Verlag, Santorini, Greece, 2011, pp. 39–49.

[26] J. L. Traff, Implementing the mpi process topology mechanism, in: Supercomputing, ACM/IEEE 2002 Conference, IEEE, 2002, pp. 28–28.

[27] C. Bordage, E. Jeannot, Process Affinity, Metrics and Impact on Performance: an Empirical Study, in: 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2018, p. 11.

[28] G. Karypis, V. Kumar, Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices.

[29] T. Hatazaki, Rank reordering strategy for mpi topology creation functions, in: V. Alexandrov, J. Dongarra (Eds.), Recent Advances in Parallel Virtual Machine and Message Passing Interface, Springer Berlin Heidelberg, Berlin, Heidelberg, 1998, pp. 188–195.

[30] E. Jeannot, R. Sartori, Improving MPI Application Communication Time with an Introspection Monitoring Library, in: 21st IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC 2020). In Conjuction with IPDPS 2020, New Orleans, Louisiana, USA, 2020, p. 10.

[31] Y. Georgiou, E. Jeannot, G. Mercier, A. Villiermet, Topology-Aware Job Mapping, The International Journal of High Performance Computing Applications 32 (1) (2018) 14–27. doi:10.1177/1094342017727061.

[32] E. Jeannot, E. Meneses, G. Mercier, F. Tessier, G. Zheng, Communication and topology-aware load balancing in charm++ with treematch, in: IEEE Cluster, Indianapolis, IN, USA, 2013, p. 8.

[33] E. Jeannot, G. Mercier, F. Tessier, Topology and affinity aware hierarchical and distributed load-balancing in Charm++, in: 1st Workshop on Optimization of Communication in HPC runtime systems (IEEE COM-HPC16), Salt-Lake City, United States, 2016.
URL https://hal.inria.fr/hal-01394748

[34] L. L. Pilla, P. O. Navaux, C. P. Ribeiro, P. Coucheney, F. Broquedis, B. Gaujal, J.-F. Mehaut, Asymptotically optimal load balancing for hierarchical multi-core systems, in: Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on, IEEE, 2012, pp. 236–243.

[35] L. L. Pilla, C. P. Ribeiro, D. Cordeiro, C. Mei, A. Bhatele, P. O. Navaux, F. Broquedis, J.-F. Méhaut, L. V. Kale, A Hierarchical Approach for Load Balancing on Parallel Multi-core Systems, in: Parallel Processing (ICPP), 2012 41st International Conference on, IEEE, 2012, pp. 118–127.

[36] I. Cores, P. Gonzalez, E. Jeannot, M. Martín, G. Rodriguez, An application-level solution for the dynamic reconfiguration of mpi applications, in: 12th International Meeting on High Performance Computing for Computational Science (VECPAR 2016), Porto, Portugal, 2016, to appear.

[37] G. Bosilca, C. Foyer, E. Jeannot, G. Mercier, G. Papauré, Online Dynamic Monitoring of MPI Communications: Scientific User and Developper Guide, Research Report RR-9038, Inria Bordeaux Sud-Ouest (Mar. 2017). URL `https://hal.inria.fr/hal-01485243`

[38] R. F. Barrett, C. T. Vaughan, M. A. Heroux, Minighost: a miniapp for exploring boundary exchange strategies using stencil computations in scientific parallel computing, Sandia National Laboratories, Tech. Rep. SAND2011-5294832.

[39] D. Bailey, T. Harris, W. Saphir, R. Van Der Wijngaart, A. Woo, M. Yarrow, The nas parallel benchmarks 2.0, Tech. rep., Technical Report NAS-95-020, NASA Ames Research Center (1995).