# Topology-Aware Job Mapping

Yiannis Georgiou, Emmanuel Jeannot, Guillaume Mercier, Adèle Villiermet

# Topology-Aware Job Mapping

Yiannis Georgiou
ATOS/Bull
Grenoble, France

Emmanuel Jeannot
Inria Bordeaux Sud-Ouest
Talence, France

Guillaume Mercier
Bordeaux INP
Talence, France

Adèle Villiermet
Inria Bordeaux Sud-Ouest
Talence, France

July 24, 2017

## Abstract

A Resource and Job Management System (RJMS) is a crucial system software part of the HPC stack. It is responsible for efficiently delivering computing power to applications in supercomputing environments. Its main intelligence relies on resource selection techniques to find the most adapted resources to schedule the users' jobs. This paper introduces a new method that takes into account the topology of the machine and the application characteristics to determine the best choice among the available nodes of the platform, based upon the network topology and taking into account the applications communication pattern. To validate our approach, we integrate this algorithm as a plugin for Slurm, a well-known and widespread RJMS. We assess our plugin with different optimization schemes by comparing with the default topology-aware Slurm algorithm, using both emulation and simulation of a large-scale platform and by carrying out experiments in a real cluster. We show that transparently taking into account a job communication pattern and the topology allows for relevant performance gains.

## 1 Introduction

Computer science is more than ever a cornerstone of scientific development, as more and more scientific fields resort to simulations in order to help refine the theories or conduct experiments that cannot be carried out in reality because their scale or their cost are prohibitive. Currently, such computing power can be delivered only by parallel architectures. Larger and larger machines are being built around the world, and being able to display such a machine has become a challenge for states and nations, scientifically as well as politically.

However, harnessing the power of a large parallel computer is no easy task, due to several factors. First, this type of computer features usually a huge amount of computing nodes and this scale has to be taken into account when developing applications. Then, the nodes architecture has become more and more complex, as the number of cores per node is in constant increase from one generation of CPU to the next. The memory hierarchy becomes also more complex, as various levels of cache are now available and the rise of MCDRAM or NVRAM will make things even more complicated in the future. Indeed, an efficient exploitation of all these types of memories is possible only if the application developer takes it into account.

One way of dealing with such complexity would be to consider the application behavior (e.g. its communication pattern, or its memory accesses pattern) and to deploy it on the computer accordingly. To this end, the most widespread technique is to determine the list of cores on which the application has to be run on, and then to bind the processes on these cores so as to minimize/maximize a predetermined criterion (a.k.a. a metric). Such a technique has already been used and investigated to improve the performance of parallel applications [13].

However, a large parallel machine is often shared by many users running their applications concurrently. In such a case, an application execution will depend on its nodes allocation, as determined by the Resource and Job Management System (RJMS). Most of the time RJMS work in a best-effort fashion, which

can lead to suboptimal allocations. That is, such allocations might be able to fulfill an application requirements in sheer terms of resources (number of CPUs, amount of memory) but might also fail to provide an environment tailored for an optimized execution. For instance, if the application processes communicate a lot between themselves, a set of nodes physically allocated apart from the rest might degrade performance severely. Furthermore, even if the given allocation is contiguous, taking into account process affinity leads to even better performance.

As a consequence, our goal is to apply to resource management the same technique that has proved its efficiency for applications deployment and execution, that is, taking into account an application's behaviour in the process of reserving and allocating the needed resources (computing nodes). This means more criteria to be used and considered by the RJMS when a user submits its request to the system. Actually, taking in account an application behaviour when allocating nodes pushes even further the idea of using an application information to improve its execution.

The contribution of this paper is the following. We detail the improvements we made to an existing RJMS in order to enable it to select the most suitable set of nodes for a given parallel application. To this end, we extend our TreeMatch algorithm and integrate it in the Slurm software to improve its ability to match the resources to the actual application communication pattern. We validate experimentally through emulation and simulation and show that the proposed solution is able to outperform the standard way of allocating nodes to resources for several different metrics. This paper is organized as follows: Section 2 gives an overview of the context and background of this work. Section 3 introduces all the software leveraged by this work before giving more technical insights about our topology-aware job allocation policy. Section 4 then shows and discusses the results obtained while related work is listed in Section 5. Finally, Section 6 concludes this paper.

# 2 Issues of Resource Allocation in Parallel Computers

## 2.1 The Sharing of Resources

A large parallel computer is to some extent a tool that has to be exploited and used. A reason as why these computers increase in size and scale stems from the fact that a substantial amount of applications grow accordingly. Therefore, an adequate platform has to match these needs. However, most of the time, such a large platform not only works in a time-sharing mode, but also in a space-sharing mode. Indeed, in order to exploit the hardware in a satisfactory way, several users share it, leading to a potentially very large number of users. An interactive access is therefore out of the question. To this end, the users have to submit their requests in terms of resources to a system called the Resource and Job Manager System (sometimes called a Batch Scheduler for short). This system's goals are threefold: 1. to centralize and analyze all the received requests, 2. to allocate the most relevant type of resources (CPU, memory or network switches for instance) able to fulfill these demands and 3. to execute the application (a.k.a. the job) submitted by a user on the set of selected resources.

There are many and sometimes conflicting criteria that should be optimized by the RJMS. Then the question that pertains to this selection and allocation of resources is to choose one. For instance, one metric could be the system throughput, that is, the amount of jobs executed during a defined time step, whilst another could be the use (CPU load) of the system. All these metrics are relevant and which to use/optimize depends on a given point of view. That is, an administrator's point of view might diverge from a user's point of view. Indeed the users hardly possess a global view of the system (in most of cases) as opposed to the administrators, hence the discrepancy.

## 2.2 Finding an Optimization Criterion

In this work, we focus on a metric relevant for users: the flow time (or turnaround time) that is, the time his/her job remains in the system. We believe it to be the most appealing one for a user seeking to gather results and get the outcome of his/her application as soon as possible. The question that now arises is how to speed up an application execution? Let us suppose that the developer has already optimized his/her application as much as it is possible. What are the means left to even speed things further up? One answer lies in the ecosystem of the application, that is, in the way the application is deployed and executed. In a previous work, unrelated to resource management and job scheduling, we showed that by taking into account an application behaviour when deploying it on the various processing entities

(CPUs, cores, threads, etc.), it is possible to improve its global execution time [14, 17]. Actually, our goal is to improve the way an application accesses its data. This data locality can be improved in several ways, but we chose so far to use the communication pattern of the application, that is, an expression of the amount of bytes/messages exchanged by the application processes. Then, we try to match this pattern to the underlying architecture by following the principle that the more processes are communicating with the others, the closer the cores they should be bound to. This can be done by several techniques but usually involves process binding and rank reordering [18].

However, the execution still depends on the set of resources allocated to the application by the RJMS. Since no guarantee is given that this allocation will be compliant with the application communication pattern, some negative side effects may occur. For instance, a subset of nodes might be physically far from another subset, thus impacting the communication between processes belonging to each subset. As a consequence, an allocation that takes into account an application communication scheme leads to performance improvements. To that end, we consider a well-known and widespread RJMS called Slurm and design a new plugin based on the TreeMatch algorithm. So far, TreeMatch was used to compute a matching between the application processes and the physical cores available. In this case we use it to determine a nodes allocation before deploying the application.

Hence, to improve the flow time we aim at reducing the job execution time of the submitted application by improving its mapping.

## 2.3 A Motivating Example

The goal of this work is to apply mapping techniques (e.g. TreeMatch) before the execution of the application processes and compare different approaches. We assume that the communication pattern of the application is known at submission time. Such a communication pattern can be gathered with application monitoring (see Section 3.1.2) or by analyzing the structure of the parallel algorithm (for instance if we are dealing with a stencil code we know which processes are communicating together and the amount of exchanged data). In any case, we assume that this communication pattern remains unchanged from one run to the other. It is not the case for all parallel applications but a large amount of applications comply to these models (for instance, dense linear applications and kernels, stencil codes, regular mesh partitioning based applications, etc. ). When the communication pattern changes from one run to an other, the proposed solution is not applicable and the user has to fall back to a standard allocation scheme: mixing the proposed topology-aware mapping with other types of mapping is fully acceptable.

| Proc. | 0 -1 | 2 - 3 | 4-5 | 6-7 |
|---|---|---|---|---|
| 0-1 | 0 | **20** | 0 | **2000** |
| 2-3 | **20** | 0 | **1000** | 0 |
| 4-5 | 0 | **1000** | 0 | **10** |
| 6-7 | **2000** | 0 | **10** | 0 |

Table 1: Affinity matrix for 8 processes (4 groups of 2 processes each). Shows the amount of bytes/messages exchanged by the application processes

Several possibilities are available. The most obvious one is to *not use* TreeMatch at all and let the Slurm environment deal with the topology by itself. The second possibility is to apply TreeMatch *just before* the job execution, once Slurm has selected the resources. Another possibility is to use TreeMatch *inside* the selection mechanism of Slurm.

An example of the difference between these approaches is depicted by Fig. 1. Let us suppose that we have 6 nodes composed of two computing entities each. We assume that node n3 is not available as computing entities 6 and 7 are already used by an other application, hence unavailable for a job allocation. Let us assume that a newly submitted job requests 4 nodes. For the sake of simplicity, we group processes in pairs (0-1, 2-3, etc.) and hence each pair of processes shall be assigned to a single node. The affinity matrix is given in table 1.

If Slurm has to allocate resources for these 8 processes, it will look for the smallest number of switches able to fulfill the request. In this case, it will require to use the whole tree. Then, it will allocate processes from left to right inside nodes in a round-robin fashion. It will allocate nodes 0, 1, 2 and 4 for the job and then map processes onto the computing entities. We can see that such an allocation is rather costly communication-wise as groups of processes are spread onto the entities and no optimization is enforced in this regard. It is therefore possible to call TreeMatch (see Section 3.1.3) to optimize the process mapping on these entities accordingly to the affinity matrix. By doing so, the resulting mapping is: group 0-1 on n0, group 6-7 on n1, group 2-3

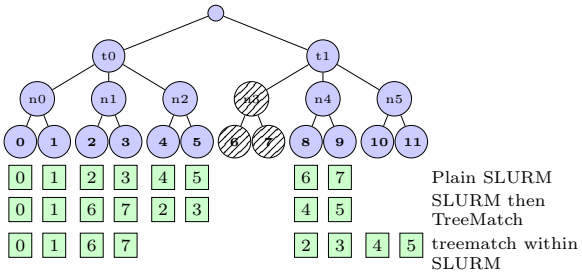



Figure 1: Tree topology of 6 nodes of 2 processing units with one unavailable nodes: n3

on n2 and 4-5 on n4. This is the best possible solution once the resources have been allocated. However, group 2-3 communicates a lot with group 4-5. With such an allocation, all the communications will transit through the root of the topology, a costly solution in terms of hops. However, a better outcome is achievable if TreeMatch performs the resource allocation. Given such a topology and the above affinity matrix, TreeMatch will allocate group 0-1 on n0, group 6-7 on n1, group 2-3 on n4 and 4-5 on n5 since there are constraints on node n3.

In this case, all the communication between group 2-3 and group 4-5 will take only 2 hops instead of 4 and therefore the communication cost is even more reduced.

# 3 A Topology-Aware Resource and Job Management System

## 3.1 Software

In this section, we detail the various software elements that we use to implement the work described in this paper. First, we describe Slurm, our target RJMS. Then, we explain the method employed to gather information about the application communication scheme (a.k.a. our *affinity matrix*). Last, we give more specific information about the TreeMatch algorithm and the constraints mapping extension we have implemented.

### 3.1.1 SLURM

We did implement a new topology-aware placement algorithm within the open-source resource and job management system Slurm [29]. Slurm performs workload management on six of the ten most powerfull computers in the world as listed by the Top500[1], including the top 1 system, Tianhe-2, which features 3,120,000 computing cores.

Slurm is specifically designed for the scalability requirements of state-of-the-art supercomputers. It is based upon a centralized server daemon, `slurmctld` also known as the controller, which communicates with client daemons `slurmd` running on each computing node. Users can request the controller for resources to execute interactive or batch applications, referred to as jobs. The controller dispatches the jobs on the available resources, whether full nodes or partial nodes, according to a configurable set of rules. The Slurm controller also features a modular architecture composed of plugins responsible for different actions and tasks such as: job prioritization, resources selection, task placement or accounting.

The resource selection process within Slurm takes place as part of the global job scheduling procedure. In particular, this procedure makes use of the `plugin/select`, which is responsible for allocating the computing resources to the jobs. Other plugins are used to facilitate and extend this procedure such as `plugin/topology` which takes into account the network topology of the cluster, the `plugin/gres` which can extend the allocation to different generic resources and the `plugin/task` which provides the isolation and possible binding of tasks on the resources.

There are various resource selection plugins within Slurm that can take into account the specificities of the underlying platforms' architecture such as `linear` and `cons_res`. The `select/linear` plugin allows the allocation of complete nodes for jobs, using simple and scalable best-fit algorithms, however, the smallest allocatable unit is the node which is quite limiting in the case of new multicore and manycore architectures. The `select/cons_res` plugin is ideal for this type of architectures where nodes are viewed as collections of consumable resources (such as cores and memory). In this plugin, nodes can be used in an exclusive or in a shared mode where a job may allocate its own set of resources differently than other jobs using the same node. The algorithms within the `cons_res` plugin are also scalable, featuring best-fit placement of jobs but they are more complex than `select/linear` since a finer granularity of allocatable resources is taken into account. One of the first version of the `select/cons_res` plugin is described in [2].

[1] `http://top500.org/lists/2015/11/`

Our studies and developments as described in the following sections are based upon the `select/cons_res` plugin. The internal representation of resources and availabilities within SLURM is made by using bitmap data structures. In the case of the linear plugin, only a node bitmap is needed whereas in the case of the `cons_res` plugin, a core bitmap is used besides the node bitmap to represent internal node resources availabilities. Within the `cons_res` plugin, the usage of node and core bitmaps is leveraged efficiently (e.g. kept separated in different contexts) in order to keep a high scalability for the selection algorithms. Another functionality of the `cons_res` plugin is the distribution of tasks within the allocated resources, which is an important feature for the optimal performance of parallel applications.

SLURM provides configuration options to make the resources selection network topology-aware through the activation of the topology plugin (`topology/tree` plugin). A particular file describing the network topology is needed and the job placement algorithms favor the choice of groups of nodes that are connected by the same network switch. The goal of the SLURM topology-aware placement algorithms is to minimize the number of switches used for the job and provide a best-fit selection of resources based on the network design. This feature becomes mandatory in the case of pruned butterfly networks where no direct communication exists between all the nodes. We use this plugin in our experiments. The scalability and efficiency of topology-aware resource selection of SLURM has been evaluated in [11].

Finally since the `cons_res` plugin deals with multicore architectures the isolation and binding of tasks on the used resources is an important feature to guarantee minimal interference between jobs sharing nodes. This feature takes place through the usage of the `task/affinity` or the `task/cgroup` plugin which use linux kernel mechanisms such as cgroups and cpusets or APIs such as hwloc [6] in order to provide the described isolation and binding.

#### 3.1.2 Application Monitoring

For this work we need to model an application communication scheme. The way communications occur describes the affinity between application processes. For the affinity matrix, we gather the communication pattern thanks to a dynamic monitoring component we did integrate within Open MPI as an MCA (Modular Component Architecture) framework called pml (point-to-point management layer). This component [5], when activated at launch time, monitors all the communications at the lowest level in the Open MPI stack (i.e. once collective communications have been decomposed into point-to-point operations). Therefore, as opposed to the standard MPI profiling interface (e.g. `PMPI`) approach where the MPI calls are intercepted, we monitor in our case the actual point-to-point communications that are issued by Open MPI, which is much more precise: for instance, we can see the tree used for aggregating values in a `MPI_Gather` call.

Internally, this component uses the low-level process ids and creates an associative array to convert sender and receiver ids into ranks valid in `MPI_COMM_WORLD`. At the end of the execution, each process dumps its local view into a file and a script aggregates all the local views at a given process to get the full communication matrix. We do not gather timing information and in general such information, as it is gathered in a distributed way is only a few MB per ranks.

#### 3.1.3 TreeMatch

TREEMATCH [13, 14], is a library for performing process placement based on the topology of the machine and the communication pattern of the application, for multicore, shared memory machines as well as distributed memory machines. It computes an allocation of the processes to the processors/cores in order to minimize the communication cost of the application.

To be more specific, it takes as input a tree topology (where the leaves stand for computing resources and internal nodes correspond to switches or cache levels) and a matrix describing the graph affinity between processes. The topology information is supplied either by the RJMS or by tools such as hwloc or netloc[2]. A hierarchy is extracted from this graph so as to match the hierarchy of the topology tree. The outcome is a mapping of the processes onto the computing resources. The objective function optimized by TREEMATCH is the Hop-Byte [30], that is, the number of hops weighted by the communication cost:

$$\text{Hop-Byte}(\sigma) = \sum_{1 \leq i < j \leq n} \omega(i,j) \times d(\sigma(i), \sigma(j))$$

where $n$ is the number of processes to map, $\sigma$ is the process permutation output produced by TREEMATCH (process $i$ is mapped on computing resource

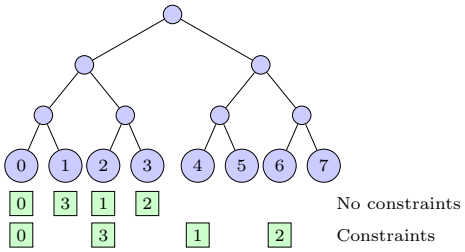
[2] `https://www.open-mpi.org/projects/netloc`

$\sigma(i))$, $A = (\omega_{i,j})\ 1 \leq i \leq n,\ 1 \leq j \leq n$ is the affinity matrix between these entities and hence $\omega(i,j)$ is the amount of data exchanged between process $i$ and process $j$ and $d(p_1, p_2)$ is the distance, in number of hops, between computing resources $p_1$ and $p_2$. In a previous work [14], we have shown that minimizing this metric allows for application runtime reduction for tree-based physical topologies.

An important feature of TreeMatch is that it only uses the structure of the tree and does not require a precise valuation of the speed of the links in the topology. Therefore, TreeMatch does not require a performance assessment of the system on which the application is going to be executed. We believe this to be a strong advantage, as gathering such information is error-prone, might be incomplete and is subject to inaccuracy.

In order to tackle the fact that not all resources are available for mapping have extended TreeMatch from [14] to take constraints into account and perform fix-vertex partitioning. When not all leaves are available for mapping (because some of them are already allocated to other applications), it is possible to restrict the leaves onto which processes can be mapped such that only a subset of the nodes is used for the mapping. To do so, we use a recursive k-partitioning algorithm where we add dummy processes that are forced to be mapped onto unavailable resources while real processes are mapped to actually available resources.

In Fig. 2, we describe an example where we map 4 processes on an architecture featuring 8 computing resources and structured as a 3-level tree. We display 2 cases: one without constraints and the other where only cores with even numbers are available for mapping.




(a) Input tree topology example.

| Proc. | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | **5** | **10** | **100** |
| 1 | **5** | 0 | **20** | **5** |
| 2 | **10** | **20** | 0 | **10** |
| 3 | **100** | **5** | **10** | 0 |

(b) Affinity matrix example.

Figure 2: Example of TreeMatch output (green square) based on the affinity matrix and the tree topology. The first line is without constraints: in this case the hop-byte metric is 360. The second line is when only cores with even numbers are allowed to execute processes (hop-bye is 660 in this case)

## 3.2 Job Allocation Strategy

We did implement a new selection option for the Slurm
`cons_res` plugin. In this case the regular best-fit algorithm used for nodes selection is replaced by TreeMatch.

To this end we need to provide three pieces of information: a job affinity matrix, the cluster topology and the constraints due to other jobs allocations.

The communication matrix is provided at job submission time through a distribution option available through the `srun` command:

```
srun -m TREEMATCH=/comm/matrix/path cmd
#SBATCH -m TREEMATCH=/comm/matrix/path.
```

Its location (path) is then stored by the Slurm controller in the data structure describing a job and can be used by TreeMatch for allocation.

As for the global cluster topology, it is provided to the controller by a new parameter in the configuration file:
`TreematchTopologyFile=/topology/file/path`.

Whenever a job allocation is computed, this topology is completed by constraints information. These constraints are provided by the nodes and cores bitmaps used by the Slurm controller to describe the cluster utilization. We need to translate this topology description into the TreeMatch topology format.

TreeMatch considers computing units as selection granularity and assign them an id considering the global topology. It must be the same for the Slurm selection plugin using TreeMatch. Hence we use the `cons_res` plugin with the configuration `SelectTypeParameters=CR_CPU` or `CR_Core`. In this case Slurm uses a cores bitmap describing precisely the location of unused CPUs inside nodes relatively to the nodes bitmap. Therefore, we need to translate Slurm local CPU ids into global TreeMatch CPU ids. Then, we use the constraints feature of TreeMatch (as described in Section 3.1) to only use CPUs not already allocated to a running job. The CPUs chosen by TreeMatch must then be translated again in new bitmaps for Slurm to use.

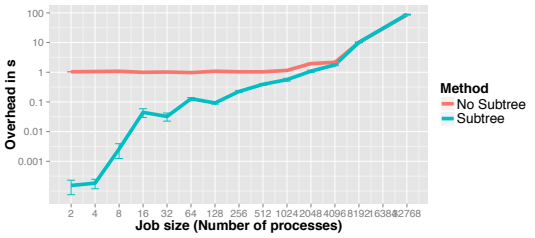



Figure 3: Comparison of TreeMatch overhead (s) for different job size on a cluster with 80640 cores between methods with and without subtrees

However, in the case of a large topology, our algorithm overhead increases: the larger the topology, the longer the TreeMatch algorithm takes. To reduce this time, we also implemented an alternative method which first finds a subtree in the global topology. Then, TreeMatch uses this subtree to rapidly choose the job allocation. To find this subtree we search through the topology tree from the leaves up to the root and from left to right. We stop as soon as we find a node with enough unused CPUs. For instance, if we consider Fig. 1 and we assume that node n0 is occupied instead of n3, then the first tree with 2 free CPUs is n1 and if we need 6 free CPUs, we shall select subtree t1.

Fig. 3 compares the overhead of this algorithm with and without the subtree optimization on a cluster featuring 80640 cores (such as the machine used in the experiments). It shows that, for jobs using less than 4096 cores, the subtree technique reduces the overhead. In any case both approaches take less than one second. At some point, the time increases linearly with the job size. However, as shown in the experiments (Section 4), the TreeMatch overhead is largely compensated by the execution time gain. Moreover, for large applications, it is possible to compute the mapping at the node level (instead of computing it at the core level): hence a full-size application (80640 cores) requires 5040 nodes which leads to an overhead of a few seconds.

For the experiments described in Section 4 we need to modify the jobs run times dynamically according to their allocation. To do this we compute for each job both the Slurm allocation and the TreeMatch one. Then we compute $R$, the ratio between their hop-byte cost (c.f. Section 3.1). We model job runtimes with computation times and communication times: $T = T_{calc} + T_{comm}$. Let $\alpha$ be the ratio of communication time of the whole runtime: $T_{comm} = \alpha T$.

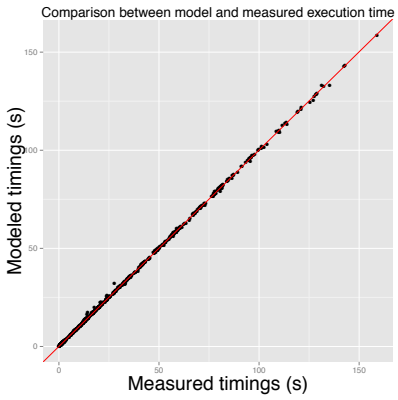



Figure 4: TreeMatch measured time vs. modeled time for the minighost application with a communication ratio between 5% and 45%

Hence, $T = \alpha T + (1-\alpha)T$. The TreeMatch impacts only the communication cost. Therefore, we model the execution time $T'$ using the TreeMatch allocation with:

$$\begin{aligned} T' &= T_{calc} + RT_{comm} = R\alpha T + (1-\alpha)T \\ &= (1 + R\alpha - \alpha)T \end{aligned}$$

We validate this model with the minighost application [3] that computes a stencil in several dimensions. We executed 84 runs with various settings (number of processors, different parameters) using a round-robin placement or a mapping computed with TreeMatch. The minighost output also provides the percentage of communication in a run. In our case, this ratio ranges from 5% to 45%. Fig. 4 shows the validation of the above model. On the x-axis is the TreeMatch runtime and on the y-axis is the predicted time based on the ratio $R$ of the hop-byte of the TreeMatch mapping and the Slurm mapping, $\alpha$ the percentage of communication and $T$ the measured execution runtime. We see a very strong (0.99993) correlation between both timings.

# 4 Experimental Validation

## 4.1 Emulation Experimental Setup

Our experiments have been carried out on the Edel cluster from the Grid'5000 Grenoble site. Edel is composed of 72 nodes featuring 2 Intel Xeon E5520 CPUs (2.27 GHz, 4 cores per CPU) and 24 GBytes of memory.

We emulate Curie (a TGCC cluster featuring 5040 nodes and 80640 cores[3]) using a SLURM internal emulation technique called `multiple-slurmd` initially described and used in [11]. SLURM uses deamons: one `slurmctld` as the controller and one `slurmd` on each node. To emulate a larger cluster, we used 16 Edel nodes and launched 315 `slurmd` daemons on each node. We can consequently submit jobs as if we were working on the Curie cluster, emulating all the job scheduling overheads. We use simple jobs (just performing a call to `sleep`) in order to provide the necessary time and space illusion to the controller that a real job is actually executing.

We base our experiments on a Curie workload trace taken from the Parallel Workload Archive[4]. We have two sets of jobs. The first set fills the cluster, and its jobs are always scheduled using SLURM in order to have the same starting point for all the experiments. The second set, called the *workload*, is the one we actually use to compare the different strategies.

All the measurements are done through the SLURM loggin system which gives us workload traces similar to the ones we obtained for Curie.

Finally, to use TREEMATCH we need to provide each job with a communication matrix. For these experiments we use randomly generated matrices featuring various sparsity rates. Indeed, the name of the application does not appear in the workload trace and therefore we cannot assess the gain an optimized mapping would yield for a given entry of the trace. However, this depends on the communication ratio of the application (the higher this ratio, the larger the possible gain in terms of runtime) and its communication pattern. Here, based on recent results [9, 12], we design the matrices such that the gain is similar to real-world applications. On average, the observed gain is 4% (resp. 11% and 18%) for a communication ratio of 10% (resp. 30% and 50%).

To evaluate our results, we use several metrics (two are for the whole workload and two are for each individual job):

- makespan: this is the time taken between the submission of the first job and the completion of the last job of the *workload*.
- utilization: this is the ratio between the CPUs used and the total number of CPUs in the cluster during the execution of the *workload*.

[3] `http://www-hpc.cea.fr/en/complexe/tgcc-curie.htm`
[4] `http://www.cs.huji.ac.il/labs/parallel/workload/`

| Com | SLURM | TM-A | TM-Isub | TM-I |
|---|---|---|---|---|
| 50% | 8318 | 6407 | 6073 | 6077 |
| 33% | 8316 | 7502 | 6821 | 6887 |

(a) Makespan

| Com | SLURM | TM-A | TM-Isub | TM-I |
|---|---|---|---|---|
| 50% | 33% | 42% | 44% | 44% |
| 33% | 33% | 36% | 40% | 39% |

(b) Utilization

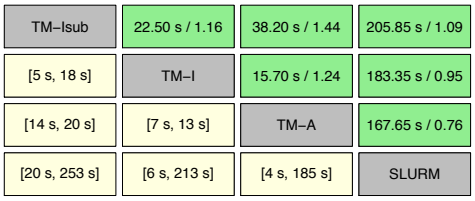
Table 2: Workload Metrics for the different strategies and different communication ratios for the emulator

- job flowtime (or turnaround time): this is the time between the submission and the completion of a given job.
- job runtime: this is the time between the start and the completion of a given job.
- job stretch: The stretch measure how the job is delayed : $f/r$ where $f$ is the flow time and $r$ the runtime. The minimum value of the stretch is 1. However, as the strategy we use changes the job runtime (it is expected to be faster with TREEMATCH than without), there is a problem in evaluating the improvement led by our proposed strategy. To ensure a fair comparison between each strategy the value of $r$ is chosen to be the runtime under the SLURM strategy.

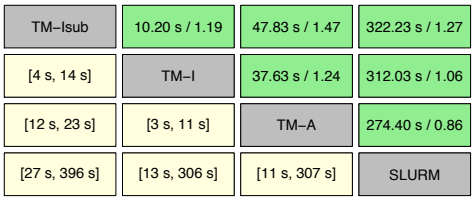
## 4.2 Emulation Results

We compare 4 cases : the classical topology-aware SLURM selection (SLURM), the same but using TREEMATCH for process placement after the allocation process and just before the execution starts (TM-A), TREEMATCH used both for the allocation process and for the process placement (TM-I) and finally the same but using the subtree technique to reduce the overhead (TM-Isub).

Here, the *workload* comprises 60 jobs. To keep the experiment duration acceptable we decrease the jobs runtimes by 50%. Table 2 describes the results obtained for this workload and two values of $\alpha$ (1/3 and 1/2). Figure 2a shows that using TREEMATCH to map the processes reduces the makespan but using it inside SLURM to allocate nodes decreases it even more. This is what is shown in Fig. 1: enhancing SLURM with TREEMATCH gives more room for optimization as the mapping and the allocation are computed at the same time.

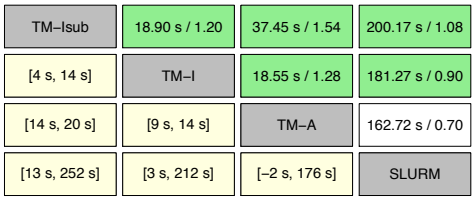



(a) 33% of communication

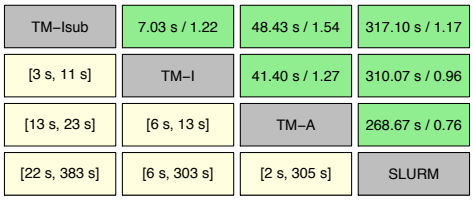



(b) 50% of communication

Figure 5: Flow time statistical comparison of methods




(a) 33% of communication




(b) 50% of communication

Figure 6: Runtime statistical comparison of methods

Moreover, the subtree optimization leads to comparable results than without the optimization. This is due to the fact that in this case, the makespan is determined by a small set of jobs and therefore the impact of this optimization is not visible for this metric. We also see that the larger the communication ratio the greater the gain: this is expected as TreeMatch optimizes only communication.

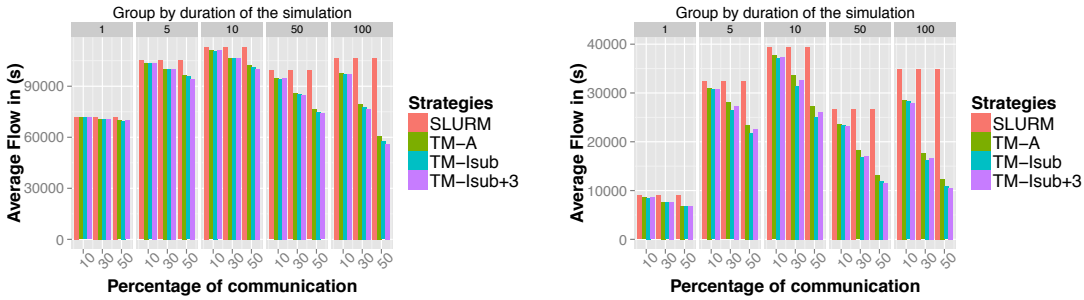
Figure 2b also shows that for the same submission workload, TreeMatch improved the resource utilization.

In Fig. 5 and Fig. 6, we use paired comparisons between different strategies for respectively jobs flowtime and jobs runtime. Here, we consider job-wise metrics, therefore we want to understand if when we average all the jobs, a strategy turns out to be better than one other. Each strategy is displayed on the diagonal. On the upper right, we have the average difference between the strategy on the column and the one on the row and the geometric mean of the ratios. For instance, in Fig. 5a, we see that on average the job flowtime is 183.35s faster with TM-I than with Slurm and the average ratio is 0.95. On the lower left part, we plot the 90% confidence interval of the corresponding mean. The interpretation is the following: if the interval is positive, then the strategy on the row is better than the strategy on the column with a 90% confidence. In this case, the corresponding mean is highlighted in green. Otherwise, we cannot statistically conclude with a 90% confidence which strategy is the best and we do not highlight the corresponding mean. For example, on Figure 5a we can see that using TreeMatch in Slurm is better than not using it. Moreover, here we see that using the subtree optimization improves the metric. For all the cases we see that TM-Isub is better than TM-I that is better than TM-A. Therefore, restricting the usage of TreeMatch improves the performance, as the gain in computing a solution overcomes the loss in terms of solution quality.

Moreover, both flowtime and runtime using TreeMatch in Slurm are shorter than using TreeMatch after Slurm, with a ratio between 1.44 and 1.54. We can also see that the more an application communicates, the smaller are the average gaps. For example, between TM-I and TM-Isub (with a 33% of communication ratio), the average difference is 22.5 s, but for a 50% ratio it is 10.2s. In these experiments, the cluster is already full when submitting the first jobs. Therefore, a part of their flowtime corresponds to the wait for a free allocation.

Figure 6 shows the comparison of jobs runtimes. We observe similar behaviors except that the confidence interval between SLURM and TM-A does not allow for a conclusion with a 90% confidence that TM-A is better than SLURM.

With these experiments we observe that using TreeMatch in the allocation process induces no negative effects whatsoever and improves the global use of a cluster. Moreover, from a user point of view, using TreeMatch can also be profitable by decreasing the runtime of his/her jobs.




(a) Full machine

(b) Empty machine

Figure 7: Average flowtime using the Curie trace with different strategies and various percentage of communication

| Com | SLURM | TM-A | TM-Isub | TM-I |
|---|---|---|---|---|
| 50% | 8317 | 6248 | 5800 | 5767 |
| 33% | 8317 | 6952 | 6671 | 6649 |

(a) Makespan

| Com | SLURM | TM-A | TM-Isub | TM-I |
|---|---|---|---|---|
| 50% | 33% | 42.8% | 46% | 45.6% |
| 33% | 33% | 39% | 40.8% | 40.5% |

(b) Utilization

Table 3: Workload Metrics for the different strategies and different amount of communication ratio for the simulator (to be compared with table 2)

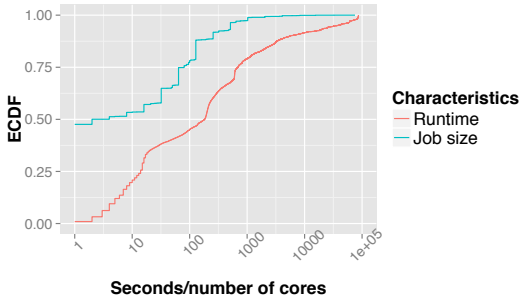



Figure 8: ECDF of runtime or jobs size of the 100 hours simulation trace

## 4.3 Simulation Results

As the experiments done in the above section are carried out through emulation, they are very long to compute (as long as the real execution times). In order to cover a larger set of test cases and a longer time-scale we have designed a simulator that simulates both the job selection part and the job execution part. Our simulator is an event-based one that reads the machine topology and the job submission trace workload and computes the start time and end time of each job based on its duration (given by the workload) and the allocation. For the job selection step, we implement the same algorithm than we use in the above section and the time to compute the allocation is based on the duration of TreeMatch when it is used or is set to 2 seconds when Slurm is used. For the execution time part, we use the formula shown in Section 4.1 if we use TreeMatch. Otherwise, the duration given by the Curie trace is used. From Table. 3 we see that our simulator is accurate enough to provide makespan duration (resp. usage) with an average absolute error of less than 3.5% (resp. 4%) and a maximum error of 7.4% (resp. 8.4%) for Table 2a (resp. 2b).

The experiments described here represent more than 12 millions node-hours. Note that allocations above 1 million node-hours are only given to application projects proposal by supercomputing centers and never to software stack optimization projects [5], hence justifying the use of simulation.

Figure 7a shows the average flow of the jobs in the case where the machine is already full of jobs (that have all been scheduled using the regular Slurm strategy). We group the measurements by simulation duration (i.e. for group 50, we consider only the jobs submitted during the first 50 hours): we go from 1 hour (365 jobs) to 100 hours (13687 jobs). For this last duration the empirical cumulated distri-

[5] See PRACE core hours award for instance: `http://www.prace-ri.eu/hermit-awardees`

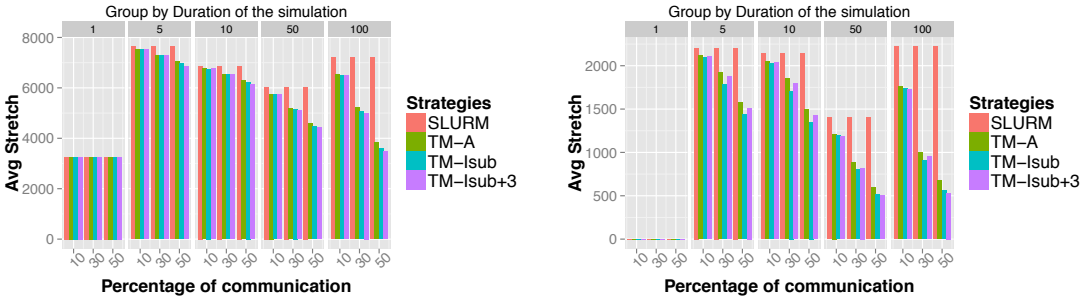
bution function (ECDF) of the job runtime (in seconds) and the job size (number of cores) is shown in Fig. 8. We see that most of the jobs uses a low number of cores on which TreeMatch as no impact while the median runtime is about 182 seconds (before applying TreeMatch). On the of x-axis Figure 7a we display the different percentage of communication (from 10% to 50%) and we have 4 strategies: the plain Slurm, TreeMatch applied at the beginning of the job to map processes to resources after Slurm has allocated the nodes (TM-A), and TreeMatch used in Slurm to compute the allocation and the mapping using the minimal subtree (TM-Isub), or 3 levels above the minimal subtree (TM-Isub+3). We see that the impact of TreeMatch on the flow increases with the duration of the simulation because at the beginning of the simulation, since the machine is full, the flowtime depends mainly on the time a job has to wait before starting, while as time goes the impact on the improvement of the mapping due to TreeMatch accumulates. Here, we do not see much difference between the different strategies involving TreeMatch because there is less room for optimization when the machine is fully utilized. However, the results are consistent, in terms of quality with the emulation results presented in the previous section. Figure 7b shows the average flow of the jobs in the case where the machine is totally empty. In this case, we see that the gain with TreeMatch increases and appears earlier which corroborates the hypothesis made in the previous paragraph. Moreover, as we have more opportunities for optimization we see that using TreeMatch in Slurm is more beneficial than using it just before the job execution. We also see a large gap for hour 5 because in the workload a large job (32768 cores) is submitted at 8461s that takes a long time to schedule and that uses a substantial part of the machine, thus impacting all the subsequent jobs.

As in the previous section, we see that even with a small average gain on each jobs (4% for the 10% communication ration case to 17% for the 50% communication ratio), we are able to achieve very large gain on the overall. This is due to the fact that these gains accumulates during the workload lifetime. We therefore expect even greater gains on real settings as the operational lifetime of a real machine is much longer than the experiments done here.

In Figure 9a, we plot the average stretch of the jobs when the machine is full or empty (Fig. 9b). We see that when the machine is empty job are executed as soon as they are submitted leading to a stretch of 1 when the simulation time is short. As for the flowtime we see that the average stretch is increasing with the simulation time. The strategies using TreeMatch within Slurm provides a better stretch and the ones using TreeMatch after Slurm. This is especially the case when the machine is empty than when the machine is full as in the former case, the strategy has more opportunity for optimization.

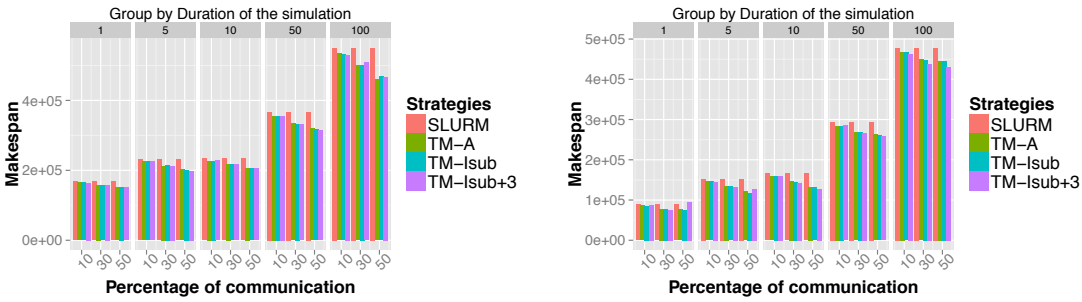
In Fig. 10 and 11 we plot the makespan and the utilization given by the simulation. We see that there is no accumulation gain for the makespan because this metrics mainly depends on the last job submission time, which is independent of the mapping strategy. For the utilization we see that, on long simulation time, we have almost no gain or even a decrease of this metric. This is due to the fact that job execution is shorter using TreeMatch leading to a lower usage of the whole machine. However, this is beneficial in the case where not all the jobs can benefit from this approach as they can use the resources freed by the jobs mapped with the TreeMatch-based strategies. Indeed, for the full machine case, with short simulation time when not all jobs have an optimized mapping we see that using TreeMatch improves the utilization (this is also consistent with the emulation experiments shown in table 2).

Last, we have also tested the case where not all the jobs are using the proposed solution (e.g. their communication pattern is not available). These applications are scheduled as any the other applications but the pattern is not taken into account to map them. Only the standard batch scheduler algorithm is used (assigning the job to the best sub-tree). They have a slight influence on the other jobs as their runtime is greater than the what it could be, but this do not change the input of the other jobs. In this case, surprisingly, the jobs that do not use the proposed solution have their flowtime reduced. Indeed, if only 50% of jobs use TM-Isub, the flowtime of the jobs using he plain SLURM strategy is reduced by resp. 5%, 13%, 25% when the percentage of communication is resp. 10%, 30% and 50%. This is due to the fact that the jobs using TM-I have a reduced runtime which lowers all jobs waiting time. This means that even if not all the jobs can use our solution, they all benefit from it.



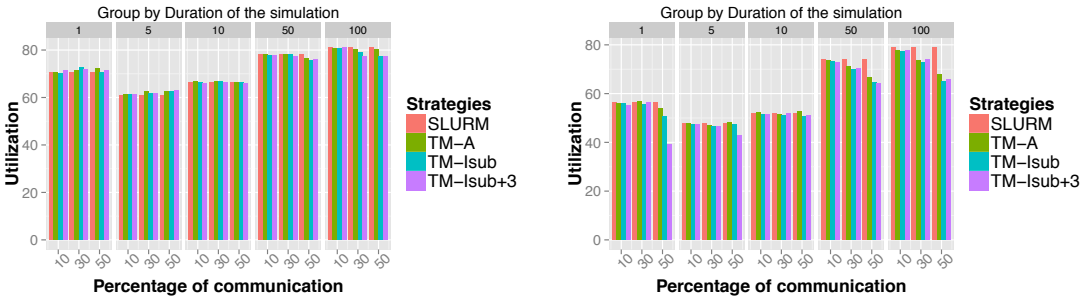

(a) Full machine (b) Empty machine

Figure 9: Average stretch using the Curie trace with different strategies and various percentage of communication




(a) Full machine (b) Empty machine

Figure 10: Makespan using the Curie trace with different strategies and various ratios of communication




(a) Full machine (b) Empty machine

Figure 11: Utilization using the Curie trace with different strategies and various ratios of communication

# 5 Related works and Discussion

The idea of using the most adequate hardware resource to a specific application is not new and has been explored in previous work. It has been particularly popular in the context of grids environments ([16, 24, 26]) where it is important to select the best set of resources (clusters in this case) to use. Such work try to reduce the impact of WAN communication in grids but do not address the deeper details of the physical topology, such as NUMA effects or cache hierarchy for instance.

More recently, some works have targeted a specific type of applications, that is, MapReduce-based applications. For instance, the TARA [15] uses a description of the application to allocate the resources. However, this work is tailored for a very specific class of applications and does not address hardware details.

The mapping of a parallel applications' tasks to the physical processors based on the network topology can lead to important performance improvements [4]. Network topology characteristics can be taken into account by the scheduler [19] so as to favor the choice of group of nodes that are placed on the same network level, connected under the same network switch or even placed close to each other so as to avoid long distance communications. This kind of feature is taken into account by most of open-source and proprietary RJMSs. However even if most of them use the characteristics of the underlying physical topology, they eventually fail to take into consideration the application behaviour when allocating resources and this is something that this work specifically addresses. HTCondor (formerly Condor) leverages a so-called *matchmaking* approach [22] that allows it to match the applications needs to the available hardware resources. However, the application *behaviour* is not part of this matchmaking and HTCondor targets both clusters and networks of workstations. SLURM [29], as previously described, provides an option to minimize the number of network switches used in the allocation, so as to reduce the communication costs during the application execution (switches that are the deeper in the tree topology are supposed to be the less costly than upper ones). The same idea of topology-aware placement is exploited by PBS Pro [21], Grid Engine[20], and LSF [25]. Fujitsu [10] provides the same but only for its proprietary Tofu network. As far as our knowledge, SLURM [29] remains the only one providing a *best-fit* topology-aware selection whereas the others propose *first-fit* algorithms.

Some other RJMS offer task placement options that can enforce a clever placement of the application processes. That is the case of Torque [8] which proposes a NUMA-aware job task placement. OAR [7] uses a flexible hierarchical representation of resources which offers the possibility to place the application processes upon the hierarchy within the computing node. However, in these existing works, only the network topology is taken in account and the nodes internal architecture is left unaddressed when performance gains are expected from exploiting the memory hierarchy.

Jingjin Wu et al. in [27] introduced a hierarchical task mapping strategy for modern supercomputers based on generic recursive algorithms for both fat-tree and torus network topologies showing very good performance with low overhead. Rashti et al. [23] proposed a weighted graph model for the whole physical topology of the computing system, including both the inter and intra node topologies. Even if both previous related works have shown interesting results with application sets, they have not been integrated with real resource and job management system neither tested with real workload traces which is our case in this paper.

A study for torus network topology [1] showed how processor ordering takes place based on space filling curve, such as Hilbert Curve, to map the nodes of the torus onto a 1-dimensional list in order to preserve locality information. This paper described the study about the allocation strategies implemented on the proprietary Cray Application Level Placement Scheduler (ALPS). Similar strategies, have been recently incorporated within SLURM with[6] (or without[7]) the use of ALPS. Another interesting work [28] adapted only for torus topology, presented a window-based locality-aware job scheduling strategy that tries to optimize job and system performance in the same time. Its goal is to preserve node contiguity by considering multiple jobs for scheduling while making use of the 0-1 Multiple Knapsack problem for resource allocation. The last two related works do not consider communication patterns as parameters in the algorithms.

Several binding policies are available, and they are compatible with the policies implemented in Open MPI. In all these solutions, the user has to retrieve

[6] http://slurm.schedmd.com/cray_alps.html
[7] http://slurm.schedmd.com/cray.html

the architectural details before submitting his/her job. Also, the placement options offered leave the user with the burden to determine his/her policy beforehand, and the application communication scheme is not taken into account.

In our case, we improve this functioning on three levels: first, we take into account not only the network but also the node internal structure. The information used is based on the *structure* of the nodes and the memory hierarchy. In other words, we do not use latency and bandwidth figures to compute our allocation. Then, this information is retrieved directly by our plugin does not have to be supplied by the user. All the technical details are hidden. Last, but not least, we also take into account not only the architecture but also the application behaviour both for the allocation and the execution of a job.

# 6 Conclusions

Job scheduling plays a crucial role in cluster administration, enabling both better response time and resource usage. In this paper, we tackle the problem of allocating and mapping jobs according to the topology and application process affinity. We extend TreeMatch to design a new allocation policy that allocates and maps at the same time application processes on the resources, based on the communication matrix of the considered application. Such strategy is implemented in the Slurm `cons_res` plugin. We tested this strategy on emulation and simulation and compare it with the standard Slurm topology-aware policy and the method consisting in mapping processes after the allocation is computed.

Results show that taking into account application characteristics and the topology provides better makespan, flow time and job runtime compared to the standard topology-aware and compact Slurm policy. On long runs, the utilization is lower but allows to accommodate jobs that do not benefit from an optimized mapping. We also show that the level at which we consider the topology impacts the performance. It is better to have a more local view of the topology than only a global view since in this latter case, allocation quality is slightly better but longer to compute. Last, even if not all the jobs are able to use this strategy all of them benefit from it with a reduced flowtime.

For future work, we would like to investigate the following research axes. First, we would like to look at fragmentation metrics. Indeed, the way jobs are allocated impacts the global resource usage and this aspect should be quantified. Also, we would like to find means to gather in a systematic fashion applications communication patterns in order to create an applications classification based on these patterns and then implement this solution in production. We would also like to validate this approach in other job scheduler such as OAR [7]. Last, we are currently working on the inclusion of our new developments in the next official Slurm release.

# 7 Acknowledgments

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr). Part of this work is also supported by the ANR MOEBUS project ANR-13-INFR-0001. This work is partially funded under the ITEA3 COLOC project #13024.

# References

[1] C. Albing, N. Troullier, S. Whalen, R. Olson, J. Glenski, H. Pritchard, and H. Mills. Scalable node allocation for improved performance in regular and anisotropic 3d torus supercomputers. In *EuroMPI 2011, Santorini, Greece*, pages 61–70, 2011.

[2] S. M. Balle and D. J. Palermo. Enhancing an open source resource manager with multi-core/multi-threaded support. In *JSSPP 2007, Seattle, WA, USA*, pages 37–50, 2007.

[3] R. F. Barrett, C. T. Vaughan, and 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*, 2011.

[4] A. Bhatele, E. J. Bohm, and L. V. Kalé. Topology aware task mapping techniques: an api and case study. In *PPOPP*, pages 301–302, 2009.

[5] George Bosilca, Clément Foyer, Emmanuel Jeannot, Guillaume Mercier, and Guillaume Papauré. Online dynamic monitoring of mpi communication. In *23rd International European*

*Conference on Parallel and Distributed Computing (EuroPar)*, page 12, Santiago de Compostella, aug 2017.

[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 *PDP 2010*, Pisa, Italia, Feb. 2010.

[7] N. Capit, G. Da Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounié, P. Neyron, and O. Richard. A batch scheduler with high level components. In *Cluster computing and Grid 2005 (CCGrid05)*, Cardiff, United Kingdom, 2005. IEEE.

[8] A. computing. Torque resource manager. http://docs.adaptivecomputing.com/torque/6-0-0/Content/topics/torque/2-jobs/monitoringJobs.htm.

[9] E. H. Cruz, M. Diener, L. L. Pilla, and P. O. Navaux. An efficient algorithm for communication-based task mapping. In *PDP'2015*, pages 207–214. IEEE, 2015.

[10] Fujitsu. Interconnect topology-aware resource assignment. http://www.fujitsu.com/global/Images/technical-computing-suite-bp-sc12.pdf.

[11] Y. Georgiou and M. Hautreux. Evaluating scalability and efficiency of the resource and job management system on large HPC clusters. In *JSSPP 2012, Shanghai, China*, pages 134–156, 2012.

[12] T. Hoefler and M. Snir. Generic Topology Mapping Strategies for Large-Scale Parallel Architectures. In *ICS*, pages 75–84, 2011.

[13] E. Jeannot and G. Mercier. Near-optimal placement of mpi processes on hierarchical numa architectures. *Euro-Par 2010-Parallel Processing*, pages 199–210, 2010.

[14] E. Jeannot, G. Mercier, and F. Tessier. Process Placement in Multicore Clusters: Algorithmic Issues and Practical Techniques. *IEEE Trans. Parallel Distrib. Syst.*, 25(4):993–1002, 2014.

[15] G. Lee, N. Tolia, P. Ranganathan, and R. H. Katz. Topology-aware resource allocation for data-intensive workloads. In *APSys '10*, pages 1–6, 2010.

[16] C. Liu, L. Yang, I. Foster, and D. Angulo. Design and evaluation of a resource selection framework for grid applications. In *HPDC '02*, pages 63–, 2002.

[17] G. Mercier and J. Clet-Ortega. Towards an Efficient Process Placement Policy for MPI Applications in Multicore Environments. In *EuroPVM/MPI*, pages 104–115, Espoo, Finland, Sept. 2009.

[18] G. Mercier and E. Jeannot. Improving MPI Applications Performance on Multicore Clusters with Rank Reordering. In *EuroMPI*, pages 39–49, Santorini, Greece, Sept. 2011.

[19] J. Navaridas, J. Miguel-Alonso, F. J. Ridruejo, and W. Denzel. Reducing complexity in tree-like computer interconnection networks. *Parallel Computing*, 36(2-3):71–85, 2010.

[20] Oracle Grid engine http://www.univa.com/oracle

[21] PBSWorks. http://www.pbsworks.com

[22] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *HPDC'7*, Chicago, IL, July 1998.

[23] M. J. Rashti, J. Green, P. Balaji, A. Afsahi, and W. Gropp. Multi-core and network aware MPI topology functions. In *EuroMPI 2011, Santorini, Greece*, pages 50–60, 2011.

[24] C. A. Santos, A. Sahai, X. Zhu, D. Beyer, V. Machiraju, and S. Singhal. *DSOM 2004, Davis, CA, USA, November 15-17*, chapter Policy-Based Resource Assignment in Utility Computing Environments, pages 100–111. 2004.

[25] C. Smith, B. McMillan, and I. Lumb. Topology aware scheduling in the lsf distributed resource manager. In *Proceedings of the Cray User Group Meeting*, 2001.

[26] O. Sonmez, H. Mohamed, and D. Epema. Communication-aware job placement policies for the koala grid scheduler. In *e-Science'06*, pages 79–86, Dec 2006.

[27] J. Wu, X. Xiong, and Z. Lan. Hierarchical task mapping for parallel applications on supercomputers. *The J. of Supercomputing*, 71(5):1776–1802, 2015.

[28] X. Yang, Z. Zhou, W. Tang, X. Zheng, J. Wang, and Z. Lan. Balancing job performance with system performance via locality-aware scheduling on torus-connected systems. In *Cluster'2014*, pages 140–148, 2014.

[29] A. Yoo, M. Jette, and M. Grondona. Slurm: Simple linux utility for resource management. In *Job Scheduling Strategies for Parallel Processing*, pages 44–60. 2003.

[30] H. Yu, I.-H. Chung, and J. Moreira. Topology mapping for blue gene/l supercomputer. In *Supercomputing'06*, New York, NY, USA, 2006. ACM.