# Topology-aware resource management for HPC applications

Yiannis Georgiou, Emmanuel Jeannot, Guillaume Mercier, Adèle Villiermet


# Topology-aware resource management for HPC applications

**Yiannis Georgiou, Emmanuel Jeannot, Guillaume Mercier, Adèle Villiermet**

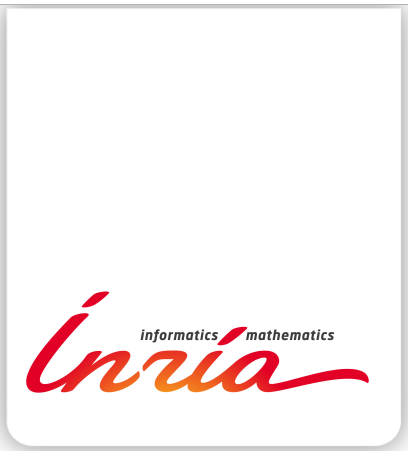
**RESEARCH REPORT**

**N° 8859**

February 2016

Project-Team Tadaam

ISSN 0249-6399 ISRN INRIA/RR--8859--FR+ENG




# Topology-aware resource management for HPC applications

Yiannis Georgiou*, Emmanuel Jeannot†, Guillaume Mercier‡,

Adèle Villiermet§

Project-Team Tadaam

Research Report n° 8859 — February 2016 — 17 pages

**Abstract:** The 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. Improper resource selection operations may lead to poor performance executions and global system utilization along with increase of system fragmentation and jobs starvation. These phenomenas play a role in the increase of the platforms' total cost of ownership and should be minimized. This paper introduces a new topology-aware resource selection algorithm to determine the best choice among the available nodes of the platform based upon their position within the network and taking into account the applications communication matrix. To validate our approach, we integrated this algorithm as a plugin for SLURM, a popular and widespread HPC resource and job management system (RJMS). We validated our plugin with different optimization schemes by comparing with the default SLURM algorithm using both emulation of a large-scale platform, and by carrying out experiments in a real cluster.

**Key-words:** resource management, job allocation, topology-aware placement, scheduling, SLURM

* ATOS/Bull
† Inria/LaBRI
‡ Bordeaux INP/LaBRI
§ Inria/LaBRI

RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

# Gestion des ressources pour les applications de calcul haute performance en fonction de la topologie

**Résumé :** Nous montrons comment la prise en compte de la topologie au moment de la sélection des ressources permet d'augmenter les performances des batch Scheduler

**Mots-clés :** calcul scientifique, HPC, gestionnaire de ressources, topologie, placement de processus

# 1 Introduction

Computer science is more than ever a cornerstone of scientific developement, as more and more scientific fields resort to simulations in order to help refine the theories or conduct expriments 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, both scientificaly or politicaly.

However, harnessing the power of a large parallel computer is no easy task, because of several factors. First, it features most of the time 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 memory is only possible if the programer takes it into account when developing his application.

One way of dealing with this complexity would be to take into account the application behavior (e.g its communication pattern, or its memory access 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 and then to bind the processes on these cores so as to minimize/maximze a predetermined criterion (a.k.a a metric). Such a technique has already been used and investigated to improve the performance of parallel applications [9].

However, since a parallel machine can be very large, it is often shared by many users running their applications at the same time. In such a case, an application execution will depend on a nodes allocation that has been determined by the Resources and Jobs Management System (RJMS). Most of the time, they work in a best-effort fashion, which can lead to suboptimal alloctions. That is such allocations might be able to fulfill an application requirements in terms of resources (number of CPUs, amount of memory) but might also fail to provide an environment tailored for an optimzed 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 serverly.

As a consequence, the idea would be to apply to resource management the same technique that has proved its efficiency for application deployment and execution, that is, taking into account an application's behaviour in the process of reserving and allocating the needed resources (computing nodes). That means more criteria to be used and taken in consideration 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.

In this paper, we shall detail the iprovements 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 did integrate our TreeMatch algorithm in the Slurm software to improve its ability to match its selection of resources to the actual use of the application. 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 more technical insights about the integration of TreeMatch into Slurm. Then Section 4 shows ans discusses the results obtained while related works are listed in Section 5. Finally, Section 6 concludes this paper.

# 2 Issues of Resources Allocation in Parallel Computers

## 2.1 The Sharing of Resources

A large parallel computer goal is to some extent a tool that has to be exploited and used. A reason why these computers increase in size and scale stems from the fact that some applications also grow accordingly Therefore, an adequate platform has to match these needs. However, a substantial part of the time, this 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 fashion, several users share it and this number of users can be potentially very large. An interactive access is therefore out of the question. To this end, the users have to submit theit requests in terms of resources to a system called the Resource and Job Manager System (sometimes called in short a Batch Scheduler). This system goals are threefold: 1– to centralize and analyze all the received requests, 2– to allocate in the most relevant fashion the resouces (CPU, memory or network switches for instance) able to fufill these demands and 3– to execute the application (a.k.a the job) submitted by a user on the set of selcted resources.

The question that pertains to this selection and allocation of resources process is the one of the criteria that should be optimized by the RJMS since there are many and sometimes conflicting one with another. For instance, one metric could be the system thoughput, that is, the amount of jobs executed during a defined time step, whilst one other could be the use (CPU load) of the system. All these metrics are relevant and which to use/optimze 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 focused on a metric relevant for users: the execution time of the submitted application. We believe that it is the most appealing one for a user seeking to gather results and know the outcome of his/her application as soon as possible. The question that now arise is how to speed up an application execution? Let us suppose that the developer has already optimized his 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, in the way the applicaton 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 [15, 12]. Actually, we try 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 cores they should be bound to. This can be done by several techniques but usually involves process binding and rank reordering [16].

However, the execution is still dependent of the set of resources allocated to the application by the RJMS. Since no guarantee is given that this allocation will be compliant with the communication pattern of the application, some negative side effects can occur. For instance a subset of nodes might be physically far from another subset, thus impacting the communication between processes belonging to each subsets. As a consequence, we believe that an allocation that takes into account an application communication scheme might lead to performance improvements. To that end, we considered a well-known and widespread RJMS, called Slurm and we integrated

| 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)

our TreeMatch algorithm within Slurm as a plugin. So far, TreeMatch was used to compute a matching between the application processes and the physical cores available. Now, we use it to determine a nodes allocation before deploying the application.

## 2.3 A Motivating Example

The goal of this work is to apply TreeMatch before the execution of the application processes and compare different approaches. We assume that we know the communication pattern of the application at submission time. Such a communication pattern can be gathered through 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 fits in these models (for instance, dense linear applications and kernels: LU factorization, Cholesky factorization, etc.).

Several possibilities are available. The obvious one is not to 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 and 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. **??**. Let us suppose that We have 6 nodes composed of two computing entities each. We assume that node n3 is not available and hence computing entities 6 and 7 are already used by an other application and are then unavailable for job allocation. Lest 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 one 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 subtree able to fulfill the request. In this case, it requires to use the whole tree. Then, it will allocate processes from left to right in a round-robin fashion inside nodes. 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 to optimize the process mapping on these entities. By doing so, the resulting mapping is: group 0-1 on n0, group 6-7 on n1, group 2-3 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.

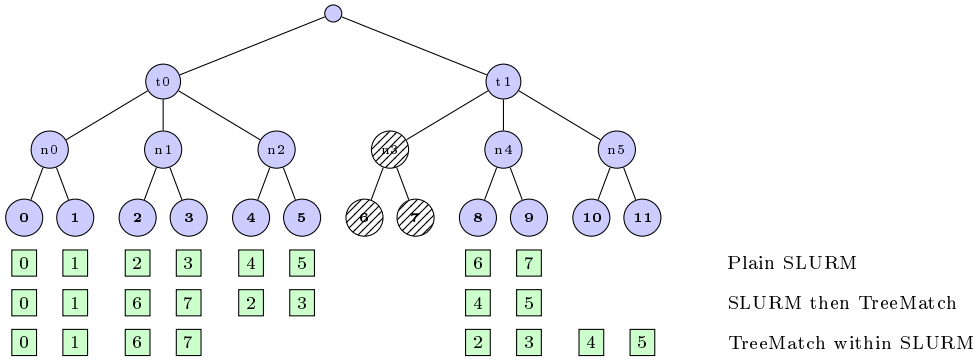



Figure 1: Tree topology of 6 nodes of 2 processing units with one unavailable nodes: n3

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 Existing software

In this section, we introduce with more details the various software elements that we used to implement the work described in this paper. First, we shall describe SLURM, our target RJMS. Then, we shall explain the method employed to gather information about the application communication scheme (a.k.a our *affinity matrix*). Then, we give more specific information about the TREEMATCH algorithm.

### 3.1.1 SLURM

We implemented the new topology-aware placement algorithm upon the open-source resource and job management system SLURM [24]. SLURM performs workload management on six of the ten most powerful computers in the world of the Top500 list[1] including the system ranked number one, 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` known as the controler, 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

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

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 lower granularity of allocatable unit is the node which is quite limiting for new multicore and manycore architectures. The `select/cons_res` plugin is ideal for this type of architectures where nodes are viewed as collection of consumable resources ( such as cores and memory). In this plugin nodes can be used exclusively or in a shared mode where a job may allocate its own resources different than other job using the same node. The algorithms within `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 versions of the `select/cons_res` plugin is described in [1].

Our studies and developments as described in the following sections are based upon the `select/cons_res` plugin therefore we try to analyze a bit more some important internals of this plugin. The internal representation of resources and availabilities within SLURM is made using bitmap data structures. In the case of linear plugin only node bitmap is needed whereas in the case of `cons_res` plugin besides the node bitmap, a core bitmap is used to represent internal node resources availabilities. Within the `cons_res` plugin the usage of node and core bitmaps is leveraged efficiently (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 group of nodes that are connected under 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 indispensable in the case of pruned butterfly networks where no direct communication exist between all the nodes. The scalability and efficiency of topology-aware resource selection of SLURM has been evaluated in [8]

Finally since `cons_res` plugin deals with multi-core architectures the isolation and binding of tasks upon the used resources is an important feature to guarantee a minimal interference between jobs sharing nodes. This feature takes place through the usage of `task/affinity` or `task/cgroup` plugins which use linux kernel mechanisms such as cgroups and cpusets or APIs such as hwloc 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 processes. To optimize the communications of a given application we therefore need to place application processes according to their affinity and the underlying physical topology. The topology information is supplied either by the RJMS or by tools such as netloc[2] and hwloc [4].

As for the affinity matrix, we gather the communication pattern thanks to a dynamic monitoring component we integrated in Open MPI as an MCA (Modular Component Architecture) framework called pml (point-to-point management layer). This component, when activated at

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

launch time (through the `mpiexec` option `--mca pml_monitoring_enable`), 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 (`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 in `MPI_COMM_WORLD`. Each time a message is sent, the sending process increments two arrays entries: the number of messages and the amount of bytes sent to the receiver. 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. This monitoring component will be released in Open MPI 2.0 and a prototype is already available on the Open MPI github platform.

#### 3.1.3 TreeMatch

TreeMatch [12] [10], 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 a permutation of the processes to the processors/cores in order to minimize the communication cost of the application.

It is also integrated in the Charm++ programming environment as an efficient load-balancer [11] and in Open MPI [16] to enable rank reordering in virtual topology management routines (e.g. `MPI_Dist_graph_create`).

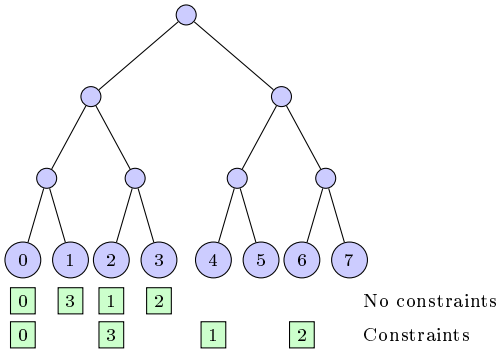
To be more specific, it takes as input a tree topology (where the leaves stand for computing resources and internal nodes correspond to switches or cache levels) and a matrix describing the graph affinity between processes. A hierarchy is extracted from this graph such that it matches the hierarchy of 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 [25], 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 $\sigma(i)$), $A = (\omega_{i,j})\ 1 \leq i \leq n,\ 1 \leq j \leq n$ 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 $p_2$.

An important feature of TreeMatch lies in its ability to take constraints into account. When not all leaves are available for mapping (because some of them are already used by other applications such as in this paper), it is possible to restrict the leaves onto which processes can be mapped such that only a subset of nodes is used for the mapping. Another 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 subject to inaccuracy.

In Fig 2, we describe an example where we map 4 processes on an architecture featuring 8 computing resources and structured as a 3-levels 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 TreeMatch Integration within SLURM

We have implemented 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 in the `srun` command:

```
srun -m TREEMATCH=/communication/matrix/path cmd
#SBATCH -m TREEMATCH=/communication/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`. With several queues we need a topology description for each of them.

Whenever a job allocation is computed, this topology is completed by constraints informations. 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.

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`. 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 (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.

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 cpus is n1 and if we need 6 cpus, we shall select subtree t1.

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

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

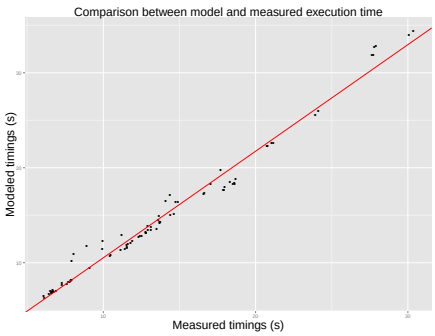



Figure 3: TREEMATCH measured time vs. modeled time for the minighost application with a communication ratio between 5% and 45%

We validated this model with the minighost application [2] that computes a stencil in various 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 ranges from 5% to 45%. Fig. 3 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 correlation between both timings even though the modeled timings tend to be slightly larger than the real ones.

# 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 with 2 Intel Xeon E5520 CPUs (2.27 GHz, 4 cores/cpu) and 24GB of memory.

We did emulate Curie (a TGCC cluster with 5040 nodes and 80640 cores[3]) using a SLURM internal emulation technique called `multiple-slurmd` initially described and used in [8]. SLURM uses deamons: one `slurmctld` as the controller and one `slurmd` on each node. To emulate a larger cluster, we use 16 Edel nodes and launch 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.

[3] `http://www-hpc.cea.fr/en/complexe/tgcc-curie.htm`

| Com | SLURM | TM-A | TM-Isub | TM-I |
|---|---|---|---|---|
| 50% | 8318 | 6407 | 9104 | 6077 |
| 33% | 8316 | 7502 | 8841 | 6887 |

(a) Makespan

| Com | SLURM | TM-A | TM-Isub | TM-I |
|---|---|---|---|---|
| 50% | 33% | 42% | 30% | 44% |
| 33% | 33% | 36% | 31% | 39% |

(b) Utilization

Figure 4: Workload Metrics for the different strategies and different amount of communication ratio

We based our experiments on a Curie workload trace taken from the Parallel Workload Archive[4]. We have two sets of jobs. the first is to fill the cluster, and the jobs belonging to this set 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 our different strategies.

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

Finally, to use TREEMATCH we need to supply a matrix communication for each job. For these experiments we use matrices randomly generated with various sparsity rates.

## 4.2 Emulation Results

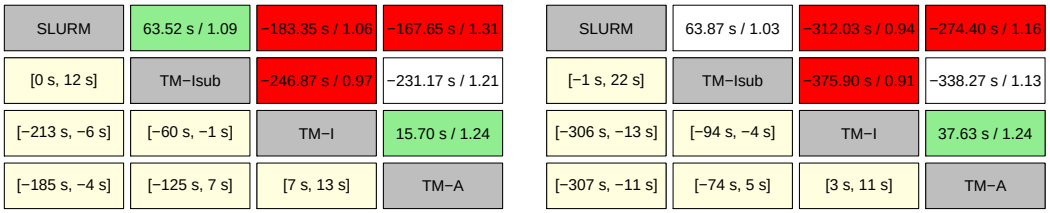
We compare 4 cases : the classical topology-aware SLURM selection (SLURM), the same but using TREEMATCH for process placement after the allocation process and just before the execution starts (TM-A), TREEMATCH used both for the allocation process and for the process placement (TM-I) and finally the same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

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 if 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*.
- job flowtime: this is the time between the submission and the completion of a given job.
- job runtime: this is the the time between the start and the completion of a given job.

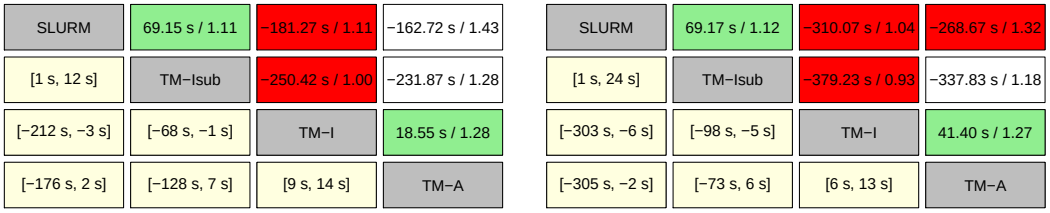
In our case, the *workload* comprises 60 jobs. To keep the duration reasonnable we decreased the jobs runtimes by a 0.5 factor. Figure 4 describes the results obtained for this workload and two values of $\alpha$ (1/3 and 1/2). Figure 4(a) shows that using TREEMATCH to reorder the process ranks reduces the makespan but using it inside SLURM to allocate nodes decreases it even more. Moreover, we proposed the subtree version to reduce the TREEMATCH overhead in the case of a large topology. However such optimization seems to be counterproductive as the results degrades as the communication percentage grows. Indeed, the TREEMATCH optimization should be more important for larger communication ratios. The explanation is the following: using the subtree optimization leaves less room for optimization and therefore can lead to an irrelevant allocation. With such a little number of jobs, a couple of bad allocations has a big impact on the whole makespan. In these cases if the applications have less communication the impact is also decreased. A solution could be to modify the subtree choice step. For instance by choosing a

[4] `http://www.cs.huji.ac.il/labs/parallel/workload/`




(a) 33% of communication (b) 50% of communication

Figure 5: Statistical comparison of selection methods: flow time



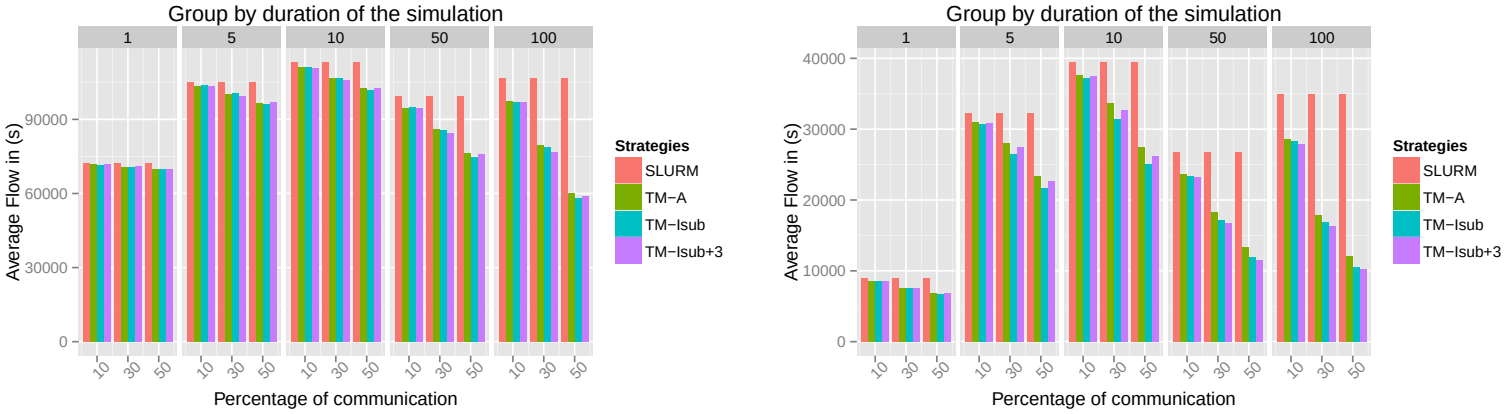

(a) 33% of communication (b) 50% of communication

Figure 6: Statistical comparison of selection methods: runtime

tree that is one level above the target subtree. This could give more choices to TreeMatch and avoid some of these performance degradations. This is tested through simulation in Section 4.3.

Figure 4(b) also shows that for the same submission workload, TreeMatch did not decrease the cluster utilization rate, except in the case of the subtree version for the same reason explained above.

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 the 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 line and the one on the row and the geometric mean of the ratios. For instance, in Fig. 5(a), we see that on average the job flowtime is 183.35s faster with TM-I than with SLURM and the average ratio is 1.06. 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 line with a 90% confidence. In this case, the corresponding mean is highlighted in green. If the interval is negative the strategy on the line is better than the one on row and the corresponding mean is highlighted in red. Otherwise, we cannot statistically conclude with a 90% confidence on which strategy is the best and we do not highlight the corresponding mean. For example, on Figure 5(a) we can see that using TreeMatch in Slurm or after is better than not using it, but using subtrees decreases the performance of individual jobs. We believe that it is because sometimes the selected subtree, due to its minimal size, leaves less optimization opportunities than the Slurm allocation. On the opposite, TreeMatch used with the whole topology information is able to compute a very good allocation and hence, the flowtime is better than with Slurm. Moreover, both flowtime




(a) Full machine

(b) Empty machine

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

and runtime using TreeMatch in Slurm are shorter than using TreeMatch after Slurm, with a ratio between 1.24 and 1.28. We can also see that the more an application communicates, the bigger are the average gaps. For example, between TM-I and TM-Isub (with a 33% of communication ratio), the average difference is 246.87 s, but for a 50% ration it is 375.90s. 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. In this case, we observe that for a 33% communication ratio, we can not compare Slurm and TreeMatch used after Slurm anymore. This means that, in this case, we gain more on the waiting time than on the execution time.

Through these experiments we observe that using TreeMatch in the allocation process induces no negative effects and can improve 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.

## 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. For the job selection step, we have implemented the same algorithm than we used in the above section and the time to compute the allocation is based on the duration of TreeMatch when it 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.

Our simulator is accurate enough to provide makespan duration with an average absolute error of less than 10% for Table 4(a).

Figure 7(a) 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 have grouped the measures by duration of the simulation (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). On the x-axis we display the different percentage of communication (from 10% to 50%) and we have 7 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 that the impact of TREEMATCH on the flow increases with the duration of the simulation because at the beginning of the simulation, as 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.

Figure 7(b) shows the average flow of the jobs in the case where the machine is totaly 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.

# 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 ([14], [23], [21]) where it is important to select the best set of resource (clusters in this case) to use. Such works 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 [13] 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 [3]. Network topology characteristics can be taken into account by the scheduler [17] 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 RJMS. However even if most of them use the characteristics of the underlying physical topology, in the end they fail to take into consideration the application behaviour when allocating resources and this is something that this work tries to address. HTCondor (formerly Condor) leverages a so-called *matchmaking* approach [20] 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 [24], 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 [19], Grid Engine[18], and LSF [22]. Os Fujitsu [7] provides the same but only for its proprietary Tofu network. As far as our knowledge, SLURM [24] 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 [6] which proposes a NUMA-aware job task placement. OAR [5] uses a flexible hierarchical representation of resources which offers the possibility to place the application processes upon the memory/cores 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.

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 the architectural details before submitting his 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 dtake 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 ou allocation. Then, this information is retrived 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 Conclusion and Future Work

Job scheduling plays a crucial role in cluster administration, enabling both better response time and resource usage. In this paper, we have tackled the problem of allocating and mapping jobs according to a cluster topology and application process affinity. We have designed 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 has been implemented in the Slurm cons_res plugin. We have tested this strategy on emulation and simulation and compared it with the standard Slurm topology-aware policy and the method consisting in mapping processes after the allocation is determined.

Results show that our solution provides better makespan, flow time, utilization and job runtime compared to these approaches and especially to the standard Slurm policy. We have also shown that the level at which we consider the topology impacts the performance in some cases. It is better to have a more global view of the topology than only a local view even if in this latter case, allocation computation time is smaller.

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.

# 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] Susanne M. Balle and Daniel J. Palermo. Enhancing an open source resource manager with multi-core/multi-threaded support. In *Job Scheduling Strategies for Parallel Processing, 13th International Workshop, JSSPP 2007, Seattle, WA, USA, June 17, 2007. Revised Papers*, pages 37–50, 2007.

[2] Richard F Barrett, Courtenay T Vaughan, and Michael 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.

[3] Abhinav Bhatele, Eric J. Bohm, and Laxmikant V. Kalé. Topology aware task mapping techniques: an api and case study. In *PPOPP*, pages 301–302, 2009.

[4] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. Hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In *Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP2010)*, Pisa, Italia, February 2010. IEEE Computer Society Press.

[5] Nicolas Capit, Georges Da Costa, Yiannis Georgiou, Guillaume Huard, Cyrille Martin, Grégory Mounié, Pierre Neyron, and Olivier Richard. A batch scheduler with high level components. In *Cluster computing and Grid 2005 (CCGrid05)*, Cardiff, United Kingdom, 2005. IEEE.

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

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

[8] Yiannis Georgiou and Matthieu Hautreux. Evaluating scalability and efficiency of the resource and job management system on large HPC clusters. In *Job Scheduling Strategies for Parallel Processing, 16th International Workshop, JSSPP 2012, Shanghai, China, May 25, 2012. Revised Selected Papers*, pages 134–156, 2012.

[9] E. Jeannot and G. Mercier. Near-Optimal Placement of MPI processes on Hierarchical NUMA Architectures. In Pasqua D'Ambra, Mario Rosario Guarracino, and Domenico Talia, editors, *Euro-Par 2010 - Parallel Processing, 16th International Euro-Par Conference*, volume 6272 of *Lecture Notes on Computer Science*, pages 199–210, Ischia Italie, SEPT 2010. Springer.

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

[11] Emmanuel Jeannot, Esteban Meneses, Guillaume Mercier, François Tessier, and Gengbin Zheng. Communication and topology-aware load balancing in charm++ with treematch. In *IEEE Cluster*, page 8, Indianapolis, IN, USA, September 2013.

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

[13] Gunho Lee, Niraj Tolia, Parthasarathy Ranganathan, and Randy H. Katz. Topology-aware resource allocation for data-intensive workloads. In *Proceedings of the First ACM Asia-pacific Workshop on Workshop on Systems*, APSys '10, pages 1–6, New York, NY, USA, 2010. ACM.

[14] Chuang Liu, Lingyun Yang, Ian Foster, and Dave Angulo. Design and evaluation of a resource selection framework for grid applications. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, HPDC '02, pages 63–, Washington, DC, USA, 2002. IEEE Computer Society.

[15] G. Mercier and J. Clet-Ortega. Towards an Efficient Process Placement Policy for MPI Applications in Multicore Environments. In *EuroPVM/MPI*, volume 5759 of *Lecture Notes in Computer Science*, pages 104–115, Espoo, Finland, September 2009. Springer.

[16] G. Mercier and E. Jeannot. Improving MPI Applications Performance on Multicore Clusters with Rank Reordering. In *EuroMPI*, volume 6960 of *Lecture Notes in Computer Science*, pages 39–49, Santorini, Greece, September 2011. Springer.

[17] Javier Navaridas, José Miguel-Alonso, Francisco Javier Ridruejo, and Wolfgang Denzel. Reducing complexity in tree-like computer interconnection networks. *Parallel Computing*, 36(2-3):71–85, 2010.

[18] Oracle. Grid engine. https://blogs.oracle.com/templedf/entry/topology_aware_scheduling.

[19] PBSWorks. Pbs. http://www.pbsworks.com/PBSProduct.aspx?n=PBS-Professional&c=Overview-and-Capabilities.

[20] Rajesh Raman, Miron Livny, and Marvin Solomon. Matchmaking: Distributed resource management for high throughput computing. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC7)*, Chicago, IL, July 1998.

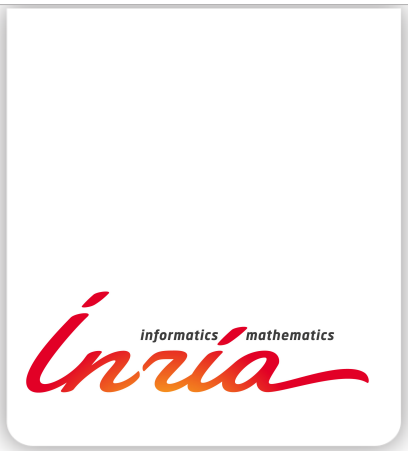
[21] Cipriano A. Santos, Akhil Sahai, Xiaoyun Zhu, Dirk Beyer, Vijay Machiraju, and Sharad Singhal. *Utility Computing: 15th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, DSOM 2004, Davis, CA, USA, November 15-17, 2004. Proceedings*, chapter Policy-Based Resource Assignment in Utility Computing Environments, pages 100–111. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[22] 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.

[23] O. Sonmez, H.H. Mohamed, and D.H.J. Epema. Communication-aware job placement policies for the koala grid scheduler. In *Proc. of the Second IEEE International Conference on e-Science and Grid Computing (e-Science'06)*, pages 79–86. IEEE Computer Science, Dec 2006.

[24] AndyB. Yoo, MorrisA. Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 2862 of *Lecture Notes in Computer Science*, pages 44–60. Springer Berlin Heidelberg, 2003.

[25] Hao Yu, I-Hsin Chung, and Jose Moreira. Topology mapping for blue gene/l supercomputer. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.




**RESEARCH CENTRE**
**BORDEAUX – SUD-OUEST**

200 avenue de la Vieille Tour
33405 Talence Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399