Design of a generic framework for resource management in *Slurm*

September 2013

Internship report
This document was writen in the context of an internship. It validates the end of my engineering school studies. This school is named ISTIA and is a part of the University of Angers. Report due time: September 2013.

Author	:	François Chevallier
Internship mentor	:	M. Yiannis Georgiou
ISTIA tutor	:	M. Mehdi Lhommeau
Course year	:	EI5 – Third year of engineering cycle – 2012/2013
Course option	:	AGI – Automation and computing engineering

Bull
Bull, Ter@tec, Bruyères-le-châtel (close to Arpajon), France

1 Preface

I want to thank *Bull* to gave me the chance to work in the HPC domain. But more particularly, I want to thank Mr. *Yiannis Georgiou*, my mentor, for all its reviews, advice, discussions and for all the help he gave me about my work, my report and my presentation.

I would like to thank Mr. *Thomas Cadeau*, my collegue, too for greatly seconding *Yiannis*. This is two great people whom I had regular contact with in *Bull*.

I want to mention and thank *Andry Razafinjatovo*, my manager, for its welcoming and its help resolving administrative problems.

I would like to thank Mr. *Matthieu Hautreux* from *CEA* for his contributions and guidance during my work and all the meetings we had together.

And I want to thank *ISTIA* to accept this internship and Mr. *Mehdi Lhommeau* for letting me be introduced to the world of HPCs and for its regular contacts and follow-up.

Contents

List of Figures

List of Tables

2 Introduction

The usage of High Performance Computing, or HPC, has many consequences on our day to day lives. We find it in medecine, financial placements, film productions, car crash simulations, carburant model usage, meteorology, aerodynamic, etc. Precise computation and complexe simulation are now possible. Thus, HPC optimizes world security, allows better world's resources utilisation, and eases world comprehension.

HPC is characterized by its calculating methods. It is established on massively parallelized computations. Parallelization at this level is not possible today on a single computer, even the most powerfull ones. It needs a lot of compute nodes (computers) organized in a sometime complex network infrastructure. HPC clusters represents a great investment. Therefore most powerfull HPC clusters can be found in the well known TOP500 list [11]. A resource and job management system is required to manage such clusters. One solution of this type is open source software *Slurm*.

Bull is an international French company acting in the HPC domain. One of *Bull*'s expertise as a company is to sell and maintain complete clustered High Performance Computing platforms. Those platforms are all integrated solutions covered by the company from hardware construction to software development, configuration and adaptation. Some of the most powerfull clusters of the TOP500 list are *Curie* and *Tera 100*, which are *Bull* platforms sold to and exploited by *CEA*. The study presented on this report took place in *Bull* company in the context of software research and development for *Slurm* Resource and Job Management System (RJMS) on HPC.

HPC clusters or supercomputers consume a lot of energy. As an example *Tera 100* can attain 5MW, an equivalent of a 5000 inhabitants town. Moreover the cluster can be not fully loaded but nodes may consume a lot of energy after all. There are a lot of possible optimisations in the domain of energy, but in other domains as well. But currently *Slurm*, as other solutions of its type, lacks needed information to make some of the possible optimisations.

On the personal side, this internship have attracted me a lot, due to its software nature. I really appreciate working on open sourced and free softwares, free as in free speech, and *Slurm* is one of these ones. *Slurm* is currently owned by *SchedMD* but has accepted a lot of contributions from the *Bull* company and the *CEA*. Sources are publicly accessible from *github.com* [4]. *Slurm* allows plugins and the basic mechanisms to develop an extensible informational framework is present.

How a new framework can be used to increase *Slurm* awareness and optimize power consumption and other characteristics of a supercomputer?

The Resource and Job Management System holds a strategic place in the HPC software stack since it keeps track of both the resources and jobs. Supercomputers become more powerfull but more complicated to manage. Resources hide information that can be taken into account for more efficient management. Information such as the placement of each resource in the actual infrastructure

can be taken into account for more efficient group shutdown of nodes. Information such as the power consumption of resources can be taken into account for power aware scheduling. Therefore a generic framework is needed to provide the means to handle those new type of information through the RJMS along with ways to retrieve them efficiently whenever they are demanded.

This report starts by a presentation of *Bull* company. Section 4 mentions related works. Section 5 provides the main part of this report and exposes the proposition of a new generic framework for resource management: the layouts framework. The following section explores the possibilities offered by the framework and its usage credibility through performance tests. The last sections provide a conclusion from the conceptual and personal point of view, along with open future perspectives.

3 *Bull*: Internship hosting company

Bull designs and develops servers and software for an open environment, integrating advanced technologies.

Throughout Europe, the Americas and Asia, *Bull*'s R&D labs are working hard today on the technologies for building the open and secure IT infrastructures of tomorrow. *Bull* has a long history of R&D success, from the first mainframe computers in the 1960s and the invention of smart cards in the 1980s, to today's new generation of Extreme Computing solutions, large-scale computing 'power plants' (Cloud computing), Open Source middleware for service-oriented IT architectures and advanced security solutions.

As a trusted IT partner, *Bull* designs, implements, maintains and runs leading-edge digital solutions that effectively combine processing power, security and the integration of complex, heterogeneous systems. With its broad portfolio of technological expertise and its positioning as an independent supplier, *Bull* works alongside customers, supporting their digital transformation, helping them to reap the full benefit and manage risk.

Bull has great expertise in the control of complex IT infrastructures, combining computing power and data management capability. Bull has been providing for many years solutions for high performance and availability, and has built strong competencies in architectures selection. This experience is a significant asset in the field of High Performance Computing, which is increasingly faced with problems in balancing capacities of calculation, I/O throughput and the hierarchical organization of storage.

Traditional *Bull* competencies on large systems were reinforced by its work in the world of Open Source and by its co-operation with scientific and technical computing actors. Moreover, *Bull* is recognized as a leader in resource management with *Slurm* to benefit from the hardware evolutions and to offer to Resources and Job Management Systems a maximum of possibilities.

Bull has already demonstrated the result of the Tera100 collaborative program with *CEA*. *Tera 100* was the first Petaflops scale supercomputer ever designed and developed in Europe. *Tera 100* has officially moved beyond the

Petaflops barrier in 2010. The efficiency and the IO performance of this supercomputer overpass all the other systems with a performance over one petaflop, a performance which makes it the most productive system in this category. *Bull* provided also to GENCI the PRACE system installed at the new computing centre (called TGCC) operated by CEA-DAM at Bruyeres-Le-Chatel. This system called *Curie* operates at 1.6 Pflops.

My internship took place in the Resource Management group with *Yiannis Georgiou* as a technical leader. This group is a part of the Middleware HPC group which has *Andry Razafinjatovo* as a manager. Middleware HPC group is part of the Inovative Products for Extreme Computing which has *Eric Monchalin* as a manager.

4 Related works

4.1 Components information base in other products

Latest evolutions in supercomputers hardware has come with an increase in power consumption, moreover, with the increased complexity in communication neworks, job placements are to be better optimized in smarter ways taking hardware components characteristics into account.

In this section, we present how other Resource and Job Management Systems (RJMS) currently have support for information on the hardware platform were they are running.

OAR [8, 1] is an open source RJMS. It has been developed in LIG laboratory. It is currently used by default on one of the biggest international-wide computer science research platform Grid5000 [5]. *OAR* use a hierarchical representation to describe resources. The cluster is treated like a tree of resources where components having a role in communication are disposed from the cluster as a root node to switch then nodes then CPUs then cores. Thus, it shows the network topology with cores as individual addressable parts. *OAR* does not have a representation of components which have no interaction with the communication layer.

OAR's tree representation can be don using a tree relational structure in a layout thanks to the new framework along with other representation. This could be used to compare existing algorithms for *OAR* with optimizations on *Slurm*.

Torque [12] is the open source version of PBSPro commercial product. This RJMS provides high availability and internal node topology scheduling.

Oracle grid engine or *OGE* [9] is an enhanced open source RJMS migrated from *CODINE* which originated from *DQS*.

Torque and *OGE* have currently no particular management of a support of information.

HTCondor [6, 10] is an innovative open source RJMS. It was developed by the Condor research project at the University of Wisconsin-Madison. It as been used as a research tool and a production system since 1984 and is still in pro-

duction. Its main advantage comes from the ability to effectively harness non-dedicated, preexisting resources under distributed ownership. Its main support of information is centered about usable resources such as nodes and their characteristics. Its representation of resources breaks cores in individual resources. They are all at the same level abstracting the node level.

Slurm [7, 13] uses plugins to add information on nodes. For example, the topology plugin add pieces of information on the network topology of the cluster. Currently only nodes information are managed since it use existant structures to support added data. There no central place or easy place in the code to place new code on.

A general remark could be done: current implementation of a base of information in current RJMS products lack multiple type of hardware components support. While they focus only on characterizing switch and compute nodes with processor, cores, memory, and now some on energy consumption, the layouts framework (section 5) propose itself as a central place to add generic components information through a generic entity model.

4.2 Job placement optimizations in other products

RJMS can take information they have to do smarter placement of jobs in order to optimize one or more characteristics. Besides, the RJMS can use information to take actions to further optimize energy consumption or other characteristics beyond just placing jobs. However, currently it seems RJMS only have algorithms to reduce energy usage by either shutting down nodes or by reducing CPU frequency of compute nodes, or algorithms to reduce communication overload according to the network topology of the cluster they run on. According to *Yiannis Georgiou's PhD section 3.3* [3], this corresponds to current main research challenges.

The tree representation of *OAR* allows it to have a very flexible and open model. It supports any kind of architectures and supports heterogeneous clusters. The scheduler use this representation to takes job demands into account.

HTCondor use their indivual resources and match their characteristics to job demands.

With its topology plugin, *Slurm* can optimize placement of jobs in order to reduce communication bottleneck. It maximize job placement on a set of nodes having the same switch. Thus communication packets does not have to go through multiple switch linked by less links than node by node link through a single switch.

Other optimization *Slurm* can provide is on energy reduction by executing a command on nodes which are not in use. Each mechanism for each optimization is implemented in discrete parts of code and plugins. The framework propose to aggregate similar part of code to provide an information base which each optimization point of view will exploit by a common way. Thus it will help start other types of optimizations. Moreover aggregating information allows future smarter optimizations by taking all pieces of information at the same time.

Generally RJMS use characteristics of nodes or CPUs or cores to let the scheduler place jobs. One other characteristic sometime taken into account is the network topology by some RJMS. But no place is currently allowed for other alteration of the scheduling strategy by other components characteristics. The new framework could allow taking multiple optimization point of views at the same time thanks to an aggregated information center.

5 Layouts framework

In this section we present the main part of this work. This is the new layouts framework and associated architectural choices taken for doing its implementation.

Slurm has support for configuration file reading and plugin creation. The layouts framework is based on this two bricks and use two other bricks implemented last year in prevision to this framework introduction: the xhash and xtree.

5.1 Introduction

Currently, RJMS in supercomputers have few or no information about their own components. The few information they have is generally about specific parts. For example, *Slurm* uses the network topology to optimize job placements.

For each new information you want to add to *Slurm*, a specific plugin must be defined for it. There is no central place to store information. However for each added plugin, there is a shared base. All plugins need to have a support to add information of components which each one is interested in. Even though being different pieces of information (power consumption, coordinates, etc), this often concerns the same set of entities.

Thus, to gather information stored in each plugin, you have to access to each plugin internal structures. This access has to be done knowing which components are shared and which one are unique.

Plugins describe relations between their components. In a rack, there is a set of nodes. So a racking plugin would want to have a parent-child relationship here. This notion can be seen as being another shared base since a normal information plugin need a support support for storage of relations.

The main goal of the layouts framework is to have a central management of two things:

- a central place to push information about cluster components: in the layouts framework we call this the **entities**;
- allowing different points of view on relations about those entities: in the layouts framework we call this the **layouts**.

Information can be related to characteristics specific to one particular "way of improvement". This can be understood as one "point of view" or "layout"

of optimization. Layouts have their own characteristics and their own point of view on entities relations. For a query a layout can give a set of entities sorted by its own definition of priority. Thus, the same query on a different layout is different. Results can be completely opposed according to layouts for the very same query.

Some query require only one final set of entities of some types at the end of the query. This may later be worked on. One solution could be to prioritizes some layout or let them put a weight on returned entities. Then, returned entities can be resorted according to the weight (or weight sum) or layout priority. Another solution could be to define an optimization problem from these priorities and to solve it using the simplex algorithm, for example.

The sets serve to take smarter decision about job placement, make power consumption optimizations, give more precise information in job parameters (a job can, for example, use maximum authorized consumption by nodes to accordingly distribute the load to them) or any other things. Utilisation examples can be found in the proof of concept, section 6.1 page 20.

By communicating with the layouts framework one can look at values associated with entities and their relations. One way of communication is proposed in the future works part, section 8. Since *Slurm* knows the global state of the cluster some live characteristics may be updated in the entities. This may be in use to let some external programs do interesting stuff with this base of information. One example is a monitoring platform doing alerts with job status, repartitions, limits, entities data and entities relations.

The new framework used to be a *Slurm* plugin at the beginning. But it has been put in its own tree to differentiate its use and implementation from other *Slurm* plugins. It is conceptually nearly different from other plugins. While other plugins are always referenced in the entire *Slurm* code base and one implementation of a plugin should be loaded, even if it is an empty implementation, layouts framework loads multiple layouts plugins. Each layout plugin can have many implementation but only one will be loaded. It is to be specified in the *Slurm* global configuration file. So we have one layout implementation loaded by layout type, but many layout types loaded at once.

Thus, the layout framework serves as an entry point for accessing all the information and entities relations brought by layout plugins. And this is the main point making layouts different from *Slurm* plugins. The **same** API is used to access all the layouts plugins.

The introduction of a new framework called "Layouts" has been discussed back in August 2012 in a CEA internship note [2] describing the main structures of layouts plugins: entities, and layout's relations.

5.2 Features

The framework features at least four things:

- easy browsing: simple browsing inside entities relations;
- fast browsing: indexed and constant time browsing, optimized access;

- quick creation of layouts: code factorization of main workflow;
- configuration file extension: extended *Slurm* parser functionalities.

The layouts framework allows easy correlation between entities thanks to their relations to other entities. These relations are defined by layouts which integrate relational structures.

Extensive usage of xhash (a hash table manager), along with xtree (a non-binary tree structure manager), took place. These structures, which were introduced in *Slurm* the last year, are completely generic and allow fast reference for xhash and fast browsing for xtree [2].

The framework is focused on the ease of future layouts additions. To that regard, it has an extremely reduced hands-on time. To create a new layout, in its simplest form, the source code file to be written must contain only some constants and functions which can be empty if not used.

In the process of aggregating functionalities to make the previous point possible, the configuration file parser of *Slurm* has been improved. It now supports full line parsing without being forced to specify a custom handling function as it was the case before. Custom handling function is still possible for handling lines, but as the new behavior covers nearly all needed cases, it is less useful. However custom handling of key-value pairs is as easy as before, and possible for line as for classic key-value pairs.

5.3 Entities

In layout framework, entities are components definitions. One entity is associated to one real hardware component or a virtual component. Virtual components can be used to do logical separations in entities relations, or in the contrary, may be used to do logical aggregation of components. Since the components do not really exist, the information on virtual entities may be interpreted differently. This must be defined in the layout plugin side.

An entity is characterized by a string identifier, a type which is defined by the layouts plugin and by its capacity to store generic information. It can store any type of data thanks to generic handling mechanism which allows memory management and dumping functions. But in order to avoid coding support for classically encountered types of data such as integers, boolean, strings, data is typed by an enumeration and support for classic types is already coded. This does not prevent custom types to be defined and used by plugins and to be handled correctly by generic management functions.

Data are associated to each entity. These data are filled up by each layout plugin. Entities are indeed shared across all layouts to enable data centralization and cross checking. Entire data concerning one entity can be easily dumped too. Thus, the data stored by one layout is accessible by another. So it is possible to have inter-dependent layouts. The entities data is the point of dependence checking.

Entities data can be anything layouts wants to store in them. However it makes more sense to store constraints. One example could be to store maxi-

mum power consumption authorized by a component to further limit the jobs consumption to never cross the limit.

A particular layout plugin will know all the keys it will use through its loading lifetime since it is its code parts that will load and store the keys in concerned entities. Currently a plugin list all its keys in the plugin definition itself. It allows the layouts manager to aggregate key names in the same place. The keys have also properties. Since the nature of the data stored referenced by a specific key will have a specific type and a specific way to be handled, the plugin can specify those ones. Currently a layout plugin can specify for a key the following things: its name, its type, the way to free the data associated to entities storing this type of data and the way to dump data associated to entities storing this type of data.

There are actually three benefit to list keys and their associated data characteristics:

- since a key will appear inside each entity a particular layout wants to fill in with particular data associated to this key, the string storing the key name is factorized in the same place, thus reducing memory usage;
- the information to deal with data where this key appears is specified one time and never duplicated;
- it allows the layouts framework to transparently add prefix to keys to avoid clashes in key naming through all layout plugins.

In figure 1, we see in the middle line keydefs which are structures loaded and held by the layouts manager from specifications given by the plugin at loading time. The structure contains the string key which will be shared. In the figure we see two entities sharing the same set of keys with different values associated to each data node.

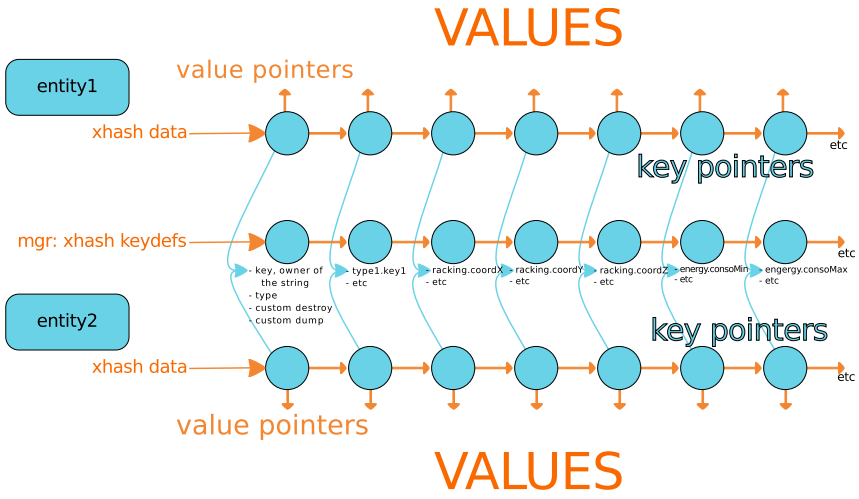

Figure 1: Entities data management

In *Slurm*, any node has a different name as unique identifier. Until now, added data was mainly node informations. To follow practices already in place and extend information concept to every components in the cluster, components names are still the unique identifier of entities.

Entities are addressed thanks to their unique names. In order to optimize access to one particular entity in the complete space of loaded entities, they are indexed. Indexation has been done thanks to a hash table: *xhash*. A hash table is a quick way to retrieve elements identified by strings (though strings are more common, other things are hashable in some other hash table implementations). Entity use strings as unique identifier. Thus, it seems to be an appropriate technical choice to use it in this context. This hash table is generic and has been mainly introduced in *Slurm* during the last year in prevision to be used for entities.

5.4 Layouts

A layout is a structure gathering:

- a name serving as a unique identifier;
- a relational structure describing relations between entities;
- other details.

At the beginning of its conception, a layout was just a tree. The main idea of a layout still applies today: it is the holder of a relational structure. Currently, layouts have only one relational structure: the tree. Nevertheless, layouts have been designed to be extensible by other types of relational structures and not only trees. The tree relational structure was sufficient to start with since it covers a lot of use cases and is simple to understand for people willing to make layouts plugins. In Future works (section 8) we suggest things that could be done as other types of relational structures such as graphs or multi-tree.

The use of tree as first relational structure is not only a technical choice. It is a design choice too. As a technical point of view it allows fast manipulation of relations. But it allows easy looking around too. Any entity can serve as an entry point. Wherever the tree is entered, one can quickly know the siblings, the parent, and children of the parametrized entity. Quickly in this context should be understand as not doing any browsing of the tree from the beginning. One can go up, up to the root of the tree, or go down, or consult siblings and go up and down as well.

As for *xhash*, a hash table, the tree structure is called *xtree* and is generic in *Slurm*. It was introduced last year for the purpose of being used by layouts in the future. Moreover, this structure is generic and can be reused wherever it is needed in *Slurm*.

A "relational node" or "node of a tree" is confusing within the context of a RJMS. Particularly in *Slurm*, in which nodes are computing servers. So from here on, in layouts or more particularly in tree context, we will call tree's nodes:

vertices. A vertex carries only one information, a data pointer. Here, this points to an entity structure. So virtually a vertex owns no information but indicates the entity which does. However, the entity itself stores every vertex pointing to it, whatever the relational structure. This is a mechanism allowing jumping around layouts through entities.

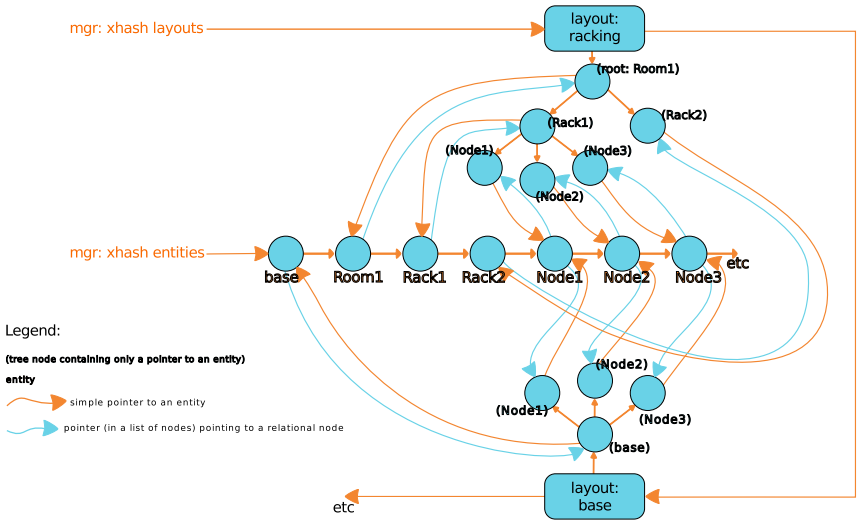

Figure 2: Example of layouts/entities instantiated relations

The figure 2 shows shared entities for two different layouts. The example loads racking and base layouts. They share three entities: *Node1* to *Node3*. Entity *base* appears only in the base layout, and *Room1*, *Rack1*, *Rack2* appear only in the racking layout. The hash table containing all entities and the one containing all layouts are both held by the layouts manager.

Layouts allow quick relational description, have fast browsing capabilities and have a small memory footprint. The last point makes it ideal for many layouts sharing the same set of entities again and again but describing different relations according to each specific point of view.

5.5 Layouts manager

The layout manager is the global structure containing instances of entities, layouts, entities data key definitions, and informations about loaded plugins. It owns the allocated memory spaces of the underlaying structures.

When a developer creates a new layout plugin, all connections to the layouts framework functions are done through the layouts manager. This is the main entry point. It is used as an API for developers of layouts.

Access to the manager's structures is private. This allows the manager to hide internal structure which is subject to change. Moreover it hides unnecessary complexity to its use. The goal here is again to have layouts users, in the rest

of the *Slurm* code this time, a simplicity of use. The manager holds details of dynamically loaded plugins. A simple user should not have the need to access this. This ensure future compatibility. Currently the layouts manager defines only two points of access:

- retrieving an entity from its name;
- retrieving a layout from its name;

As a starting point, retrieving those by names is going to be a common operation. In order to optimize it they have been placed in hash tables indexed by their unique names.

From this on, all relational vertices where an entity appears is accessible from it. And all entities appearing on a layout is accessible from it. Thus, it is possible to jump from layouts to layouts by using entities as jumping point.

Apart defining the API and being the owner of global structures, the manager has currently another role. It hosts factorized logic which is common for layouts plugins. Common part includes:

- loading plugin, storing its methods entry points, grabbing its definition;
- generating keys definitions;
- reading the layout plugin configuration file;
- generating new entities, merging existing ones;
- populating entities data;
- instantiating layouts, storing general information about it;
- building relations between entities for each layouts;

To exist, each layout has to specify two more things than basic *Slurm* plugin requirements:

- plugin specification;
- all required methods for custom behavior (for now, most can be empty).

A plugin specification is just a structure where things needed by the layouts manager are held. Currently, there are:

- the options: this specifies how to parse configuration file for what is not already managed by the layouts manager;
- the keyspec: this specifies what keys (keys names) will be possibly stored in entities by this layout plugin, type of data associated to them, and how to free and dump this associated data;
- the structure type: this specifies what type of relational structure the layout will use, currently, it is only possible to specifies tree.

- entity types: this specifies what entity types will be recognized for entities lines in plugin dedicated configuration file;
- automerge: this specifies whether to try to load automatically entity data from configuration file to entities data fields by the layouts manager or if the layouts wants to manage all this on its own with a custom callback;

Thanks to this specification the layouts manager can load, construct, and merge its main structures. This mechanism has been proposed to make the creation of a layout plugin easy. The layout plugin creator has just to focus to what logic he had to write in order to add value to loaded information or relations. A special effort has been done to avoid every layout to repeat same parts of code over and over. This specification is one of the possible answer to this problem.

The configuration parser at the beginning had not the possibility to parse a whole line on its own. It was the responsibility of the user of the parser to define callbacks appropriately. The configuration parser has been modified to allow a whole line to be parsed by using suboption. More than that, since there was a sense to have it on a whole line, the parser has gain the possibility to expand a line. This means a line can specify parameters as it has been specified multiple times with common prefix.

As an example a line like:

```
Entity=asterix[1-4]    Type=Node   CoordsZ=[1-4]
```

will expand itself to:

```
Entity=asterix1    Type=Node   CoordsZ=1
Entity=asterix2    Type=Node   CoordsZ=2
Entity=asterix3    Type=Node   CoordsZ=3
Entity=asterix4    Type=Node   CoordsZ=4
```

There are longer examples in *Slurm* configuration examples. Another point about the parser is that since the expansion was added, the merging functionality has had a sense to be added. As of now, then a master key, the most left key, is specified multiple times, either on its own or by multiple expansion, values are merged together before returning it back to the caller. If a value on the same key has been specified multiple times, the policy is to override values. The last specified value is always the one taken into account.

Figure 3 shows big lines of previous explanations. It presents the four main structures and their main fields:

- *layouts_mgr*: the main structure used by the layout manager for holding references to other structures;
- *keydef*: the structure holding definitions of keys used in entities data key-value pairs; is instantiated at the layout plugin loading time; a key definition means the main string identifying a piece of data and the way to manage it;

- *layout*: the layout structure; this one is already detailled in section 5.4;
- *entity*: the entity structure; more details about it are in section 5.3.

You will find little representations of examples of what can be found in the *tree*, *data*, and *nodes* fields. Nodes field is often reffered to as vertices in this report.

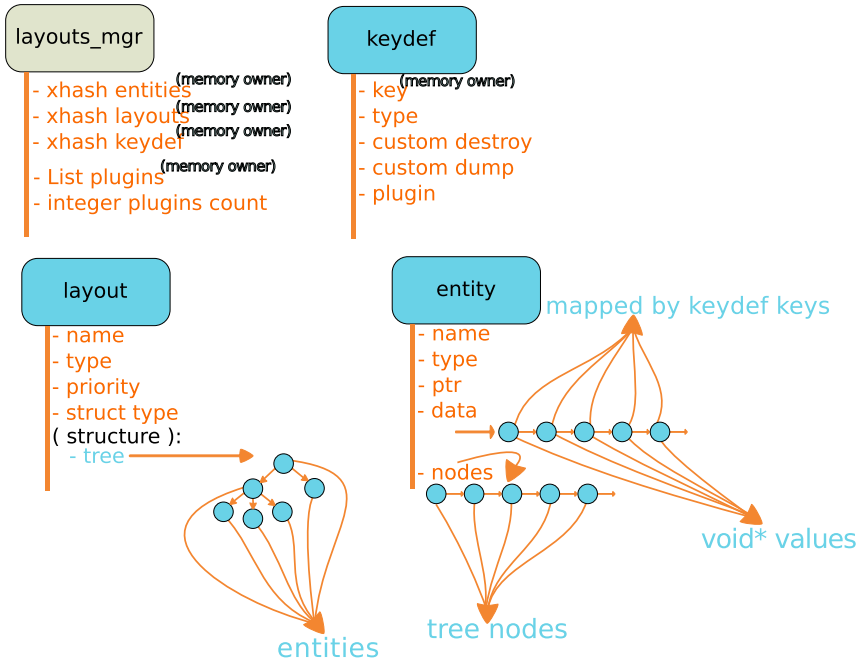

Figure 3: Main data fields of plugin structures

Figure 4 shows a timeline of the main event of the code. It helps to represent aggregated code role distribution. The top left event is the start of the main program: the *Slurm* controller, *slurmctld*. Only the *init* and configuration steps are represented since the framework is not called as for now in any other part of the code. Then main functions of the layouts manager are called. Near the beginning of the initialization of layouts manager, the layouts plugins initializations are involved collecting constants characterizing each layout. Then the layouts manager use layout and entities code to load and construct them from the configuration file of each layouts plugin.

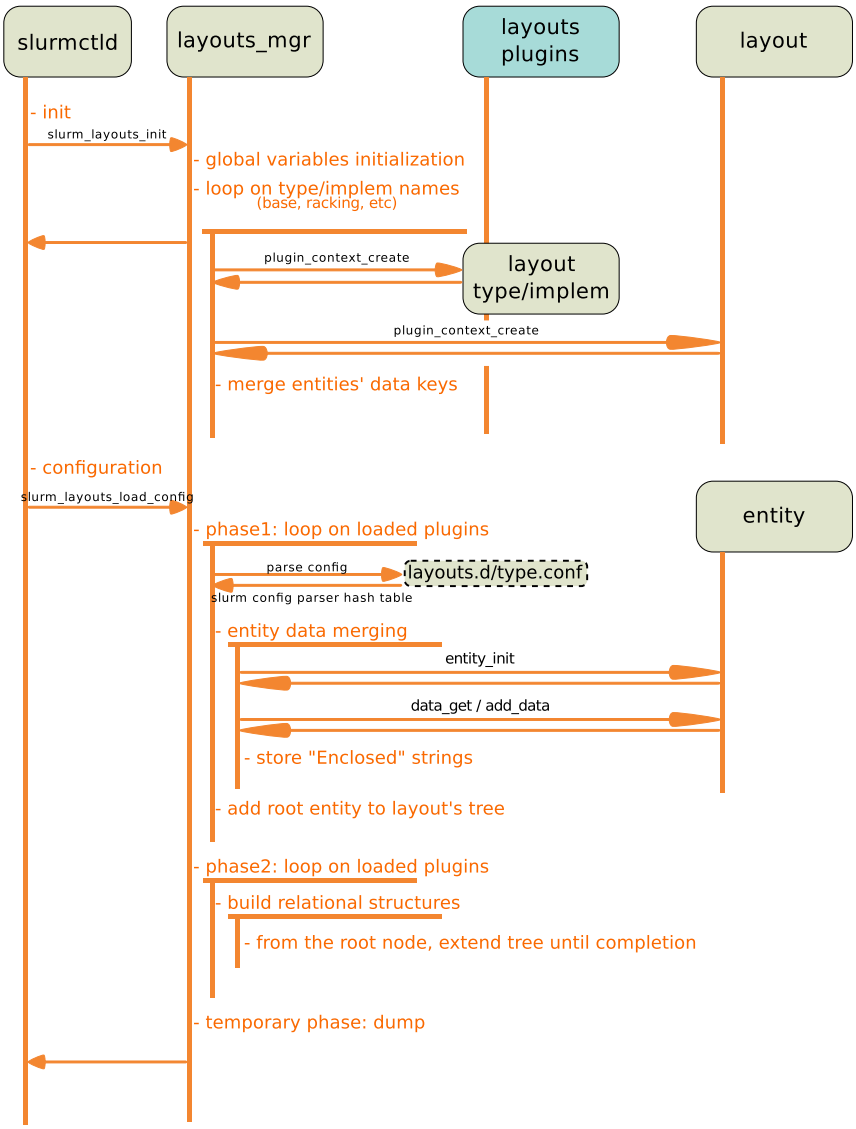

Figure 4: Timeline of code execution

The layout manager is an interface allowing developers to add its layout plugin next to others very quickly. One of its main role is to aggregate common functionalities from layouts.

5.6 Summary

This section presented the way we chose to implement a support of information for *Slurm*. It was designed from the beginning to simplify the insertion of new

information types based on a generic architecture. Furthermore, the technical programming choices were made having performance and scalability in mind. The proposition presented here allows to add and manage any information in a generic but extensible manner. Further investigations of performances of this framework and possible utilisations are done in the *Proof of concept* section (section 6).

6 Proof of concept

The introduction of layouts brings a lot of new possibilities that would not be possible with the current mechanims and features in *Slurm* or other RJMS. The layouts are mainly a support of new type of information for resource management. They enable easy plugin creation and simple addition of new information or new points of view about component's relations. This may enable future optimizations in more than one domain.

The Layouts management is not just a plugin, it is a new framework. It slightly differs from *Slurm* classical plugins. Layouts are all loaded under the same entry point. Requests are directed against all layouts, not just one. Operations are aggregated to produce a result.

6.1 Possible utilisations of layouts

One possible utilisation of the layouts would be to do a better distribution of jobs accross the entire cluster according to the maximum energy consumption authorized by entity. Distribution decisions help would come from a layout named *energy* for example. This would allow maximum limit for specific equipement and total limit on parents. *Slurm* could take actions to respect this limit by either:

- reducing the cpu frequency of some compute nodes;
- shutting down or putting to sleep compute nodes; or
- not lauching a job on power hungry compute nodes or not lauching it at all.

It is possible based on the same principle to distribute jobs according to a set of optimizations criteria. Optimization of communications according to the total link speed taking into account each components in the communication path is another way to have better job placements. These are two fundamentally different point of views with different job placement satisfying at the best each of them. According to each job requirement, or to the owner of the cluster preference, one point of view may have to be satisfied first and then the other to the extend of the possible. Or we can imagine some sort of compromise to satisfy as best as possible each layouts according to a priority property on layouts.

It is possible to have more than one way to optimize the energy than just having a maximum consumption limit that should not be crossed. It might be wanted to load already used racks (those having compute nodes with loaded jobs but having some free compute nodes as well) in order to keep some other racks completely free. This allows *Slurm* to shutdown or act at the power consumption level of not just compute nodes but accompanying equipments. But this kind of optimization can, as well as the one presented in the previous paragraph, lead to a completely different repartition of jobs across the cluster. Thus a mediator mechanism would have to be implemented in the layouts manager, taking priorities as weight and trying to refine a list of compute nodes for the scheduler to finally place jobs on, as suggested on previous paragraph. But other approaches are possible such as simplex resolution.

One completely different example of utilisation of layouts can be in the monitoring area. If an external monitoring solution, since it is not the role of *Slurm* to be a complete full featured monitoring solution, wants to know some metrics or some characteristics to raise alerts, *Slurm* can be an interesting entry point. With the layouts framework, *Slurm* knows the hardware of compute nodes and it already knows the software (jobs) to be run on them. This makes it good for a global view of the cluster activity and error gathering at the job level. If *Slurm* integrates a dumping of entities values on demand, it is possible to communicate information on external tools.

One possible scenario of an alert being raised by this system might be a job with a great importance to be placed. The maximum consumption limit has been set to a limit which *Slurm* would have taken in account. *Slurm* would have set some racks in a sleep or shutdown state to respect this limit. Then *Slurm* determine that to keep the energy consumption under the limit, it would have to either force the shutdown of some running jobs or to cross the limit to keep all jobs even with a reduction of the frequency of the CPU of all the compute nodes of the cluster (or some jobs would have better priority and would have requested a greater frequency). Then by some sort of communication, the monitoring solution would have raised an alert with some details accessible from the layouts such as the consumption limit causing the alert, and from *Slurm* the job characteristics. According to this parameters an administrator then would have all the information needed to react accordingly to either keep the new prioritized job in the queue or to kill running jobs (maybe asking the owners of the jobs before) or completely other decisions. Some monitoring solutions can take automatized actions according to the received information. Thus it would be possible to automatize some administrative actions this way too.

6.2 Performance evaluation: different clusters

In this section we present performance evaluation experiments along with our observations. The layouts framework was designed with scalability in mind and we want to be sure that the duration of loading and using layouts is as optimized as possible.

The machine used for doing tests is part of a cluster accessible from inside of

the *Bull* company. The cluster is called *mo*. Only one node was used to do the tests. No more is required since *slurmctld*, the program integrating the layouts framework, only needs one. Node *mo73* was allocated for the tests. There is no other particular requirement for the node than being idle at the time of the tests.

This particular node characteristics are:

- Processor: 16 cores, Intel(R) Xeon(R) CPU E5-2660 0 @ 2.20GHz
- Memory: 32 Gibibytes.

For each test, a configuration file is read. This serve as an input for tests performances. Each configuration file is associated to a cluster and a layout plugin. In particular, there is two of them here:

- racking.conf
- energy.conf

Each graphic in the next figures shows times in histograms. In the order, there is:

- *init*: This step loads a layout plugin, instantiate layout structure, and all associated variables.
- *phase 1*: This step loads the layout configuration file, read entities, merge them, and add the root vertex to the layout structure.
- *phase 2*: This step walks from the root vertex entity and descends to children, reading enclosed attribute in order to build a tree reading.
- *walk entities*: This step goes through all entities in the global hash table and access to all their attributes.
- *walk layouts*: This step goes through each layouts, and goes through each of their vertices in the top-bottom-left-right order and access to entities names.

The tests are done executing the *slurmctld* program, quitting as soon as the timings are done, just after the configuration reading phase of global *slurmctld*. Test are run 10000 times to avoid bad approximations due to other activities in the node.

Each table and each figure in tests of the performance evaluation section presents classic statistics functions about the result series of each phase in each test. Those functions are the average, standard deviation, variance, minimum and maximum of the series. *Numpy* package and corresponding functions (avg, std, var, min, max) in the *Python* language was used to do the computations and to generate figures. The times in the result series are found by making the difference of a call of *gettimeofday* C function at the begin and the end of each phase.

Formulas used for these functions are:

- $average = \frac{\sum_x Element_x}{Number\_Of\_Elements}$
- $variance = \frac{\sum_x (Element_x - average)^2}{Number\_Of\_Elements}$
- $standard\_deviation = \sqrt{variance}$

Following subsections present:

- the cluster *Sara* which is a real cluster with a limited number of nodes (400 nonetheless) and have its configuration extracted from an existing database describing components;
- the cluster *Curie* which is a real cluster too which have its configuration extracted from the same type of database but have a lot more nodes; and
- the cluster fictive which is a fictive cluster, but with a configuration file which is a lot cleaner since it has manually been written.

6.2.1 *Sara*

This configuration comes from a cluster called *Sara*. The configuration was automatically generated by a python script from a database describing the cluster components and parent-child relationships. It is a database specific to cluster sold by the *Bull* company. It is called *clusterdb*. The python script simply takes all information to generate a configuration file parsable by *Slurm*. The configuration file is meant to be read by racking layout plugin.

Sara has under 400 nodes.

The figure 5 clearly shows that the most time hungry step is the phase 1. This is where all parsing and structure allocations are done.

This may explain the deviation being so big to be compared to the average since it depends a lot on system activities.

But keep in mind, despite the scale between phase 1 and other step times being different, phase 1 still generally takes less than 22ms. Even the poorest case takes 0.65s. Refer to table 1 for precise numbers.

	init	**phase1**	**phase2**	**dump entities**	**dump layouts**
Average	107	21330	1478	465	211
Standard deviation	26	6893	202	69	23
Variance	696	47519196	40805	4736	521
Minimum	100	21067	1462	461	206
Maximum	1791	655345	21633	7320	2445

Table 1: *Sara* racking times (times in microseconds)

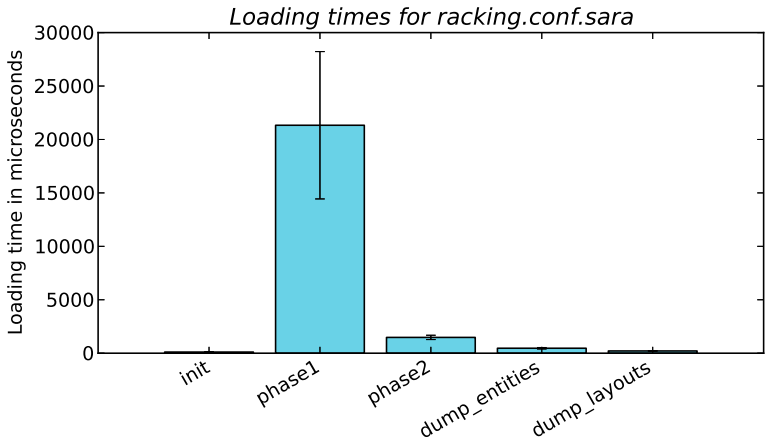

Figure 5: *Sara* racking times

6.2.2 *Curie*

As for *Sara*, this configuration comes from *clusterdb* and is meant to be read by the racking layout plugin.

In this test *Curie*, one of the biggest cluster in France, has a lot of compute nodes: around 5 thousands.

We can see in figure 6 the same tendance as for *Sara* in the graph. It is just the scale which change. We can see the timing is linked to the number of entities loaded and particularly in this case to compute nodes.

Another thing we can see is that the most affected time is phase 1. It may be in function of the number of line in the configuration file. The standard deviation is near 1/30 of the average. Numbers can be found in table 2. The step seems to be more repetable. The cause may be the file size again. Being bigger, the time speed for reading the file should be more repetable than system call for opening/closing the file. Those may be more affected by other system conditions.

	init	**phase1**	**phase2**	**dump entities**	**dump layouts**
Average	113	289280	21616	6901	2378
Standard deviation	57	8809	227	73	27
Variance	3246	77605534	51453	5261	735
Minimum	105	4978	486	560	136
Maximum	3080	833707	22454	7603	2575

Table 2: *Curie* energy times (times in microseconds)

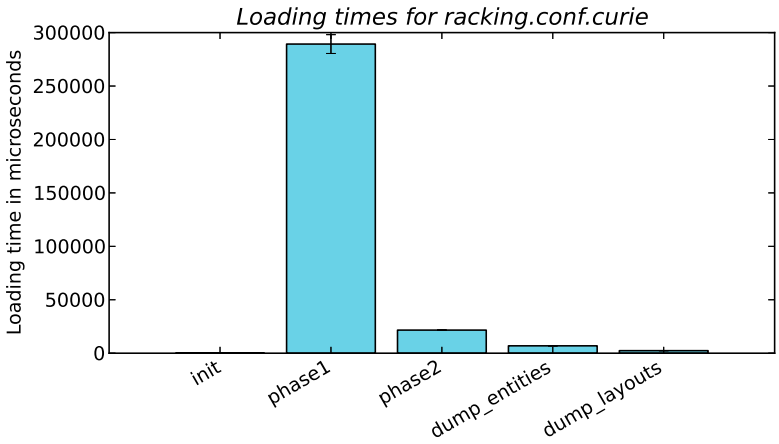

Figure 6: *Curie* energy times

6.2.3 Fictive

This configuration is a fictive configuration. It represents a non-existing cluster named *Asterix*. *Asterix* has 400 compute nodes.

Figure 7 and its corresponding table 3 describe racking layout configuration. It shows a phase 1 a little bit optimized compared to *Sara* and *Curie* clusters. This is due to a better organization in the configuration file. This one is not generated but written by hand.

	init	**phase1**	**phase2**	**dump entities**	**dump layouts**
Average	107	4788	491	525	139
Standard deviation	79	59	226	11	6
Variance	6217	3430	51128	130	42
Minimum	99	4740	483	518	135
Maximum	7953	8295	23088	1303	537

Table 3: Fictive cluster racking times (times in microseconds)

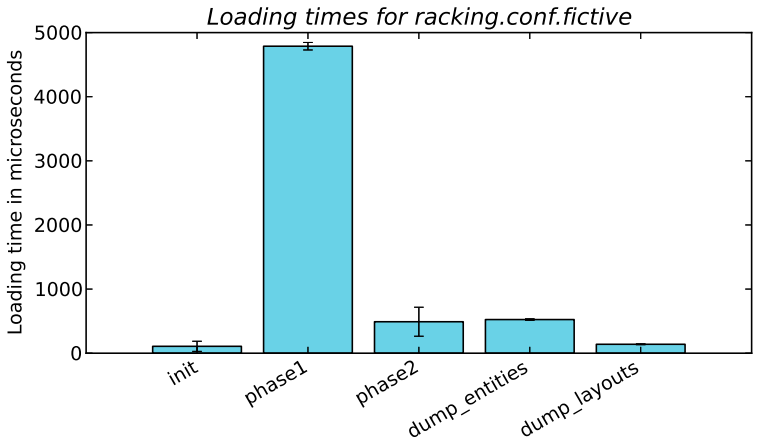

Figure 7: Fictive cluster racking times

Figure 8 and its corresponding table 4 describe energy layout configuration. Compared to racking test, phase 2 is taking more time. This is due to a more complexe tree of entities.

	init	**phase1**	**phase2**	**dump entities**	**dump layouts**
Average	114	6578	3626	1205	776
Standard deviation	42	62	29	13	10
Variance	1744	3862	816	168	95
Minimum	99	6509	3584	1189	762
Maximum	3832	10788	4034	1532	949

Table 4: Fictive cluster energy times (times in microseconds)

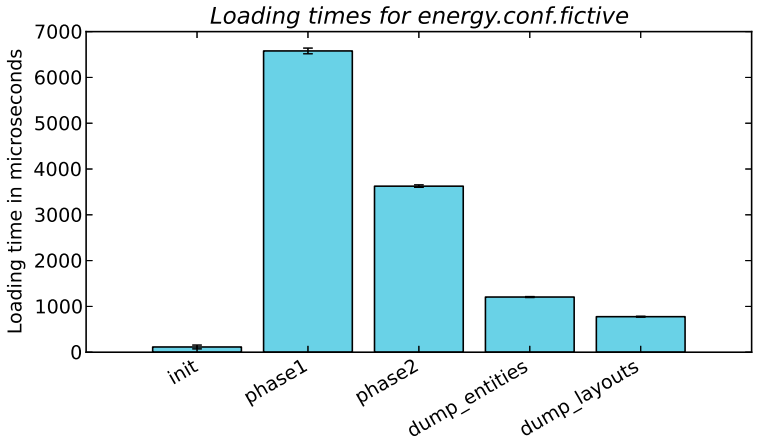

Figure 8: Fictive cluster energy times

Figure 9 and its corresponding table 5 describe both layouts configuration combined and loaded in the same instance of the *Slurm* controller. Unsurprisingly the tendance shown in the figure is the same of previous added, one on the top of the other. We can see a bigger standard deviation which should be due to the opening of two files instead of just one.

	init	**phase1**	**phase2**	**dump entities**	**dump layouts**
Average	168	10711	3952	1441	892
Standard deviation	45	1269	135	48	22
Variance	2004	1609383	18105	2309	501
Minimum	137	10610	3909	1430	881
Maximum	4595	137528	16059	6144	2990

Table 5: Fictive cluster racking and energy combined times (times in microseconds)

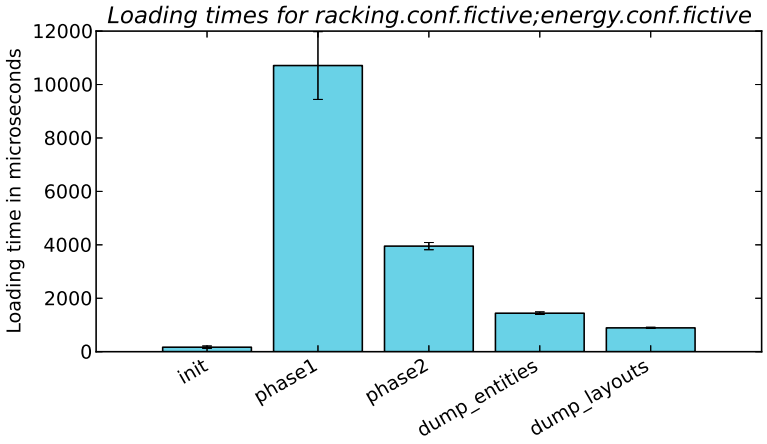

Figure 9: Fictive cluster racking and energy combined times

Using more than one layout allow us to be closer to the reality. The goal of layouts manager is to ultimately aggregate a lot of layouts with their entities in order to make a good decision to satisfy optimization of an set of characteristics, not just the racking nor just the energy.

Optimization has to be done globally and underlying speed of loading and access to multiple loaded layouts and their entities is a key factor to test. In this case two layouts loaded allows us to check if times is near the addition of the test of each layout individually.

We can imagine to do other possible tests with more layouts activated at the same time such as one for the network topology.

6.2.4 Conclusion of timing

Globally the results show the reading of the configuration file and the instantiating and converting times are the most time consuming operations.

But even the cluster with the biggest number of nodes available on the tests generally takes less than 350 milliseconds for its *phase 1*.

6.3 Performance evaluation: file size factor

Configuration file is one the two most time consuming operation in the most time consuming step. We propose in this section to study the impact of the configuration file size on loading times. Normally the dumping times should not change, we will verify it in the following figures and tables.

Fictive cluster is called *mynode* for this subsection. It is a different fictive cluster than the previous fictive *asterix* cluster. It has 10000 compute nodes composing it which is near the double of the number of nodes that composes *Curie*.

Testing with a imaginary big cluster allows us to make supposition about the scalability possibilities of the layouts framework.

	init	**phase1**	**phase2**	**dump entities**	**dump layouts**
Average	111	134961	15674	5710	2867
Standard deviation	11	4843	1665	29	21
Variance	111	23454403	2773584	820	441
Minimum	105	133819	15393	5664	2832
Maximum	799	495631	181975	6426	3239

Table 6: Configuration file complex 0 times (times in microseconds)

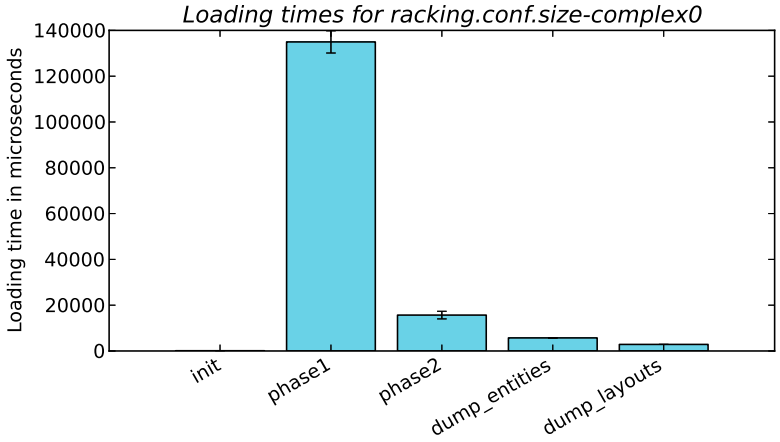

Figure 10: Configuration file complex 0 times (times in microseconds)

	init	phase1	phase2	dump entities	dump layouts
Average	112	134157	15438	5691	2859
Standard deviation	12	4371	155	29	22
Variance	135	19104139	24110	833	474
Minimum	105	133360	15196	5649	2827
Maximum	762	410410	22865	6536	3265

Table 7: Configuration file complex 1 times (times in microseconds)

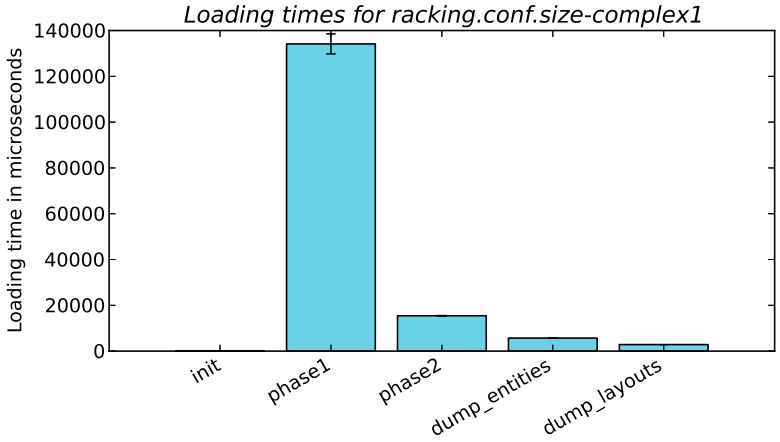

Figure 11: Configuration file complex 1 times

	init	phase1	phase2	dump entities	dump layouts
Average	112	137292	15508	5755	2859
Standard deviation	33	2092	119	6462	22
Variance	1063	4378253	14247	41751629	501
Minimum	104	136709	15362	5647	2827
Maximum	2467	343401	23182	651904	3218

Table 8: Configuration file complex 2 times (times in microseconds)

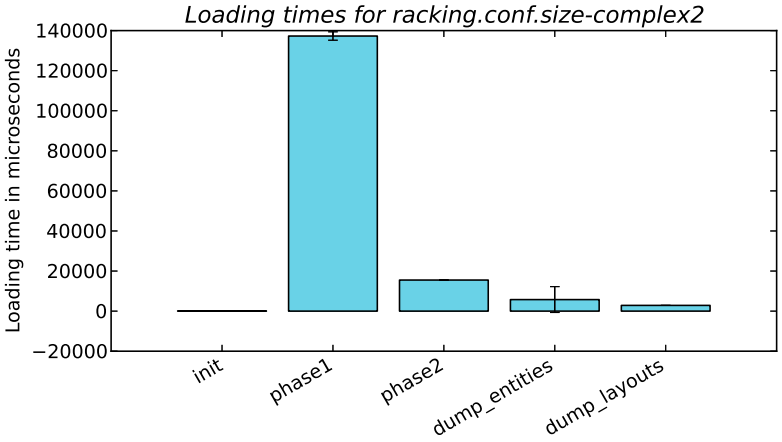

Figure 12: Configuration file complex 2 times

	init	**phase1**	**phase2**	**dump entities**	**dump layouts**
Average	112	158119	15851	5691	2862
Standard deviation	13	5501	3391	30	23
Variance	160	30262596	11496329	876	521
Minimum	105	157290	15669	5645	2829
Maximum	946	499383	354316	6973	3299

Table 9: Configuration file complex 3 times (times in microseconds)

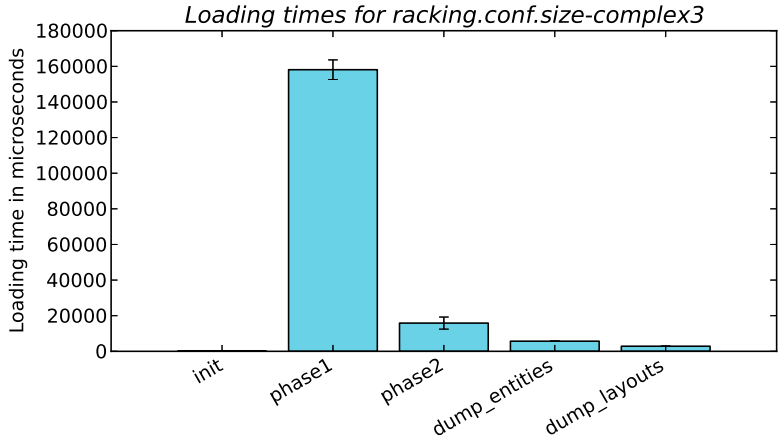

Figure 13: Configuration file complex 3 times

	init	phase1	phase2	dump entities	dump layouts
Average	112	307313	16657	5696	2867
Standard deviation	7	6508	265	88	36
Variance	45	42347841	70325	7736	1283
Minimum	105	305105	35	5651	26
Maximum	496	673130	25849	14024	3261

Table 10: Configuration file complex 4 times (times in microseconds)

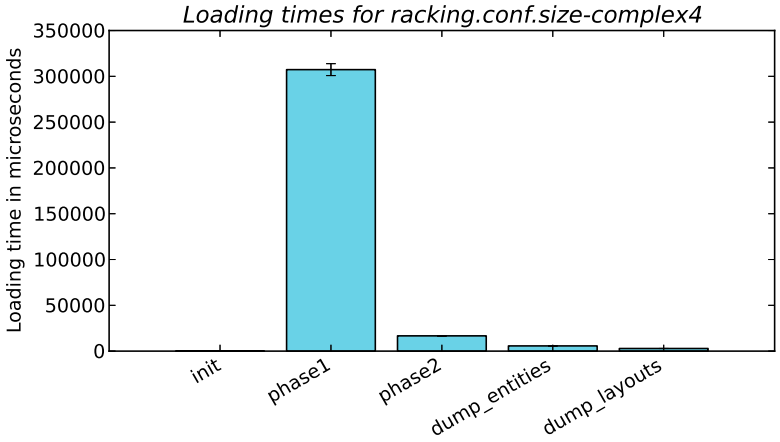

Figure 14: Configuration file complex 4 times

The figures 10, 11, 12, 13, 14, and their corresponding tables 6, 7, 8, 9, 10, show a little increase in the phase 1 times from case 0 to case 4. However those files share a common base describing the tree complex relations. Each complex level is increased by the power of 10 of expansion done manually in the file for the description of entities of node type (1, 10, 100, 1000, 10000). Complex 0 starts with all the description of nodes contained in a single expandable line. Then complex 1 has 10 expendable lines with each line describing 1000 nodes. Then power of 10 are exchanged to finish with complex 4 which has every nodes describe by a single line for each one. Thus in complex 4 test, we have 10000 lines for nodes only and the shared common base.

We can compare *Curie* and complex 4 in term of number of lines and complexity. They take nearly the same amount of time (289ms for *Curie* and 307ms for complex 4). Although complex 4 has 2 times more nodes than *Curie*, *Curie* has more containers where each containers contains only 2 to 4 nodes, bringing us to nearly the same tree complexity and the same final number of lines.

For figures 15, 16, and their corresponding tables 11, 12, in opposition to the previous tests, the configuration files describe a simpler tree, where all entities inherit from a single root entity. Configuration files describe the same set of entities and the same relations. The difference between tests is the factorized one describe in a single line all child entities, whereas expanded file describe each

entity using a whole line each time. Expansion on nodes extend node names.

	init	phase1	phase2	dump entities	dump layouts
Average	112	109904	13375	12208	2552
Standard deviation	17	6128	142	78	33
Variance	289	37550267	20051	6035	1084
Minimum	104	109068	13195	12098	2526
Maximum	1104	626944	21071	13930	3996

Table 11: Configuration file fully factorized times (times in microseconds)

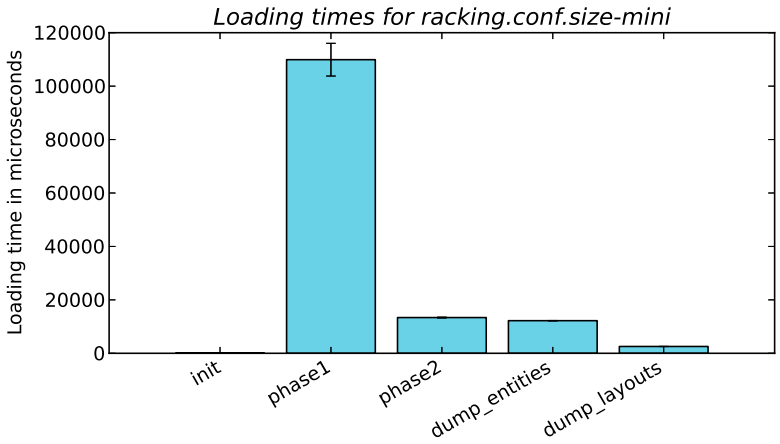

Figure 15: Configuration file fully factorized times (times in microseconds)

	init	phase1	phase2	dump entities	dump layouts
Average	112	346819	15095	12468	2554
Standard deviation	9	7868	4020	77	29
Variance	75	61899255	16158397	5886	824
Minimum	105	111347	13470	12362	2530
Maximum	617	764597	416745	14623	3978

Table 12: Configuration file with expanded nodes times (times in microseconds)

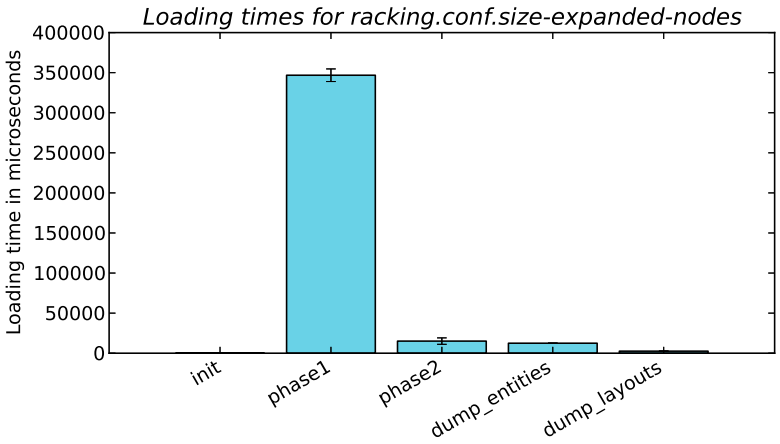

Figure 16: Configuration file with expanded nodes times

The dumping times are nearly the sames with a variation of around 500 microseconds. With an increase of 10000 lines of configuration file for the expanded nodes versus the factorized one, we see an increase of nearly 3.5 times. We generally are still under 0.35s for expanded nodes files and under 0.77s in the worst case in the tests. So we can think that even with generated files for future clusters, layouts framework can be acceptable at the controller loading time.

7 Conclusion

Today HPC evolutions requirements are such than smarter scheduling is needed in order to optimize at least energy consumption and network communications. Modern research should add other optimizations areas on top on those ones. Optimization is done on the same set of hardware components and take specific information on those to take decision on job placements or action on components themself. Thus, optimization on one area can not be done in an independant way without takin the risk to suboptimize other areas. Optimization areas, or optimization point of views, must be taken as a whole at the same time. In order to simplify regroupement and to ease the add of pieces of information, the *layouts* framework has been introduced.

Slurm has been chosen for this study since it holds a strategic place in the HPC software stack. Keeping trace of both the resources and jobs, it has a global vision of the cluster usage. *Slurm* generates and manages the cluster main activity by the distribution of computing tasks, jobs. *Slurm*, as other RJMS, lacks a central place to add information of supercomputers hardware components. The *layouts* framework is an answer to this problem.

The *layouts* framework is design to be a generic central place to store information of real and virtual hardware components. This first feature, called

entities, allows it to store all needed pieces of information whatever the form it has through generic handle mechanism of data for memory management and dumping management. The second main aspect of *layouts* framework is its capability to have multiple optimization point of view. This one, called itsel layouts, is a relationnal organization of entities. Relations are organized in function of the way *layouts* plugins authors wants to further exploit component dependencies to further optimize one characteristic.

The performance evaluations are a pratical way to validate if the contribution is usable in real case scenario. According to the tests which were done, the new *layouts* framework is acceptable. Even on benchmarking clusters with the quickest jobs which lasts one second, *layouts* framework speed shows that it is fast enough to exploit a lot of concurent optimization point of views within the execution of running jobs.

The introduction of a new framework in *Slurm* opens a whole new area in the domain of supercomputers component information. New possibilities should arise pretty quickly after inclusion in the upstream repository.

I'm really happy to contribute to such a great domain in the computing world. I hope my work will help future evolutions of the product. This internship brought new visions of massive parallel computing possibility of our information era.

Examples of future possibilities given by this framework are given in section 8.

8 Future works

Some of the utilisation examples of this new framework are already given in the section 6.1 utilisation examples. This little section focuses on possible evolutions beyond those examples.

Dynamic update of entities data is one of the most useful one. Data might be updated without restarting the controller again. The controller then might use this to reorganize and further optimize distribution of jobs and take actions according to fluctuant characteristics. One of them for example might be a maximum power consumption limit which might be updated according to energy distributor capacity by week, month or whatever.

One other possible evolution concerns the exploitation of all information gathered by *Slurm*. Since after adding a lot of layouts, given the ease to make them, *Slurm* might be the center of a base of information of the supercomputer components on which it is run on. To simplify access to information, it is possible to make a virtual file system. Thanks to *fuse* this should not be too much complicated. Thus it would possible to browse the filesystem as you would normally have done with normal file browser and to exploit information with classic tools. The vertices of a tree are easily represented by directories and data of entities by text files, thanks to custom dump functions, or, by binary files if the user knows the format of the data. This should be applicable to graphs representation too, for example and, an update procedure may be added later

on, just as the kernel does with its */proc* pseudo-filesystem.

In some network topology, the network nodes are interconnected in a complex way. For example it is possible to have multiple link to interconnect some racks to avoid congestion if they were to pass by a single central switch or router. Figure 17 shows a simplified example of such a network on the right side, the left side being a simple version representable with a tree structure.

This type of network topology is not representable in a tree. Therefore it might be wanted to introduce a graph structure as a new relational structure of layouts, since layout structure is already made to accept new ones.

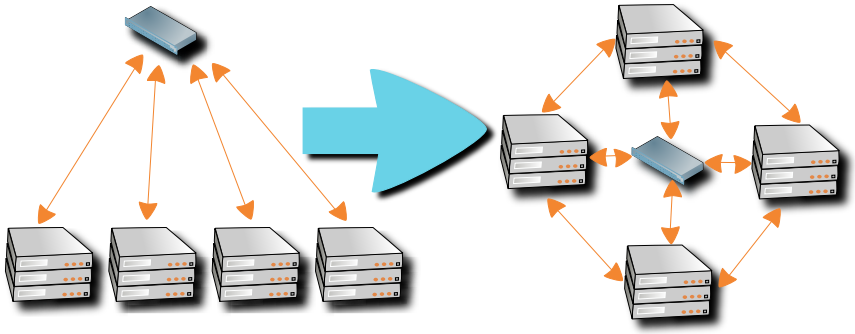

Figure 17: Big picture example of more complex network topology

One simpler type of relational structure though is a multi-tree. It can increase representation possibilities with less complexity than choosing directly the graph relational representation. One possibility quickly passed on during the internship is the possibility of using current *xhash* for multi-trees. Using a virtual root node as parent of multiple sub-trees should be possible but the usage is to be tested.

Appendices

A Conf file example

In this section, configuration files used to generate tests used to make the plots in the section 6.2, performances evaluation.

Here you find the configuration file used to make fictive racking tests.

```
Priority=10

Root=Bruyeres

Entity=Bruyeres Type=Center Enclosed=Room1

Entity=Room1 Type=Room Enclosed=Row[1-2]

Entity=Row1 Type=Row CoordsX=1 Enclosed=Rack[1-2]
Entity=Row2 Type=Row CoordsX=2 Enclosed=Rack[3-4]

Entity=Rack1 Type=Rack CoordsY=1 Enclosed=chassis[1-2]
Entity=Rack2 Type=Rack CoordsY=2 Enclosed=chassis[3-4]
Entity=Rack3 Type=Rack CoordsY=1 Enclosed=chassis[5-6]
Entity=Rack4 Type=Rack CoordsY=2 Enclosed=chassis[7-8]

Entity=chassis1 Type=Chassis CoordsY=1 Enclosed=asterix[0-49]
Entity=chassis2 Type=Chassis CoordsY=2 Enclosed=asterix[50-99]
Entity=chassis3 Type=Chassis CoordsY=1 Enclosed=asterix[100-149]
Entity=chassis4 Type=Chassis CoordsY=2 Enclosed=asterix[150-199]
Entity=chassis5 Type=Chassis CoordsY=1 Enclosed=asterix[200-249]
Entity=chassis6 Type=Chassis CoordsY=2 Enclosed=asterix[250-299]
Entity=chassis7 Type=Chassis CoordsY=1 Enclosed=asterix[300-349]
Entity=chassis8 Type=Chassis CoordsY=2 Enclosed=asterix[350-399]

Entity=asterix[0-49]    Type=Node   CoordsZ=[1-50]
Entity=asterix[50-99]   Type=Node   CoordsZ=[1-50]
Entity=asterix[100-149] Type=Node   CoordsZ=[1-50]
Entity=asterix[150-199] Type=Node   CoordsZ=[1-50]
Entity=asterix[200-249] Type=Node   CoordsZ=[1-50]
Entity=asterix[250-299] Type=Node   CoordsZ=[1-50]
Entity=asterix[300-349] Type=Node   CoordsZ=[1-50]
Entity=asterix[350-399] Type=Node   CoordsZ=[1-50]
```

Here you find the configuration file used to make fictive energy tests.

```
Priority=10
Root=Bruyeres

Entity=Bruyeres Type=Center Enclosed=Room1

Entity=Room1 Type=Room Enclosed=Row[1,2]

Entity=Row1 Type=Row Enclosed=Rack[1-2]
Entity=Row2 Type=Row Enclosed=Rack[3-4]

Entity=Rack1 Type=Rack ConsoMIN=100 ConsoMED=110 ConsoMAX=150 Enclosed=chassis[1-2],switch1,rackdoor1
Entity=Rack2 Type=Rack ConsoMIN=100 ConsoMED=110 ConsoMAX=150 Enclosed=chassis[3-4],switch2,rackdoor2
Entity=Rack3 Type=Rack ConsoMIN=100 ConsoMED=110 ConsoMAX=150 Enclosed=chassis[5-6],switch3,rackdoor3
Entity=Rack4 Type=Rack ConsoMIN=100 ConsoMED=110 ConsoMAX=150 Enclosed=chassis[7-8],switch4,rackdoor4

Entity=switch[1-4]   Type=Switch    ConsoMIN=3  ConsoMED=15 ConsoMAX=45
Entity=rackdoor[1-4] Type=RackDoor  ConsoMIN=40 ConsoMED=50 ConsoMAX=60

Entity=chassis1 Type=Chassis ConsoMIN=10 ConsoMED=40 ConsoMAX=50 Enclosed=asterix[0-49]
Entity=chassis2 Type=Chassis ConsoMIN=10 ConsoMED=40 ConsoMAX=50 Enclosed=asterix[50-99]
Entity=chassis3 Type=Chassis ConsoMIN=10 ConsoMED=40 ConsoMAX=50 Enclosed=asterix[100-149]
Entity=chassis4 Type=Chassis ConsoMIN=10 ConsoMED=40 ConsoMAX=50 Enclosed=asterix[150-199]
Entity=chassis5 Type=Chassis ConsoMIN=10 ConsoMED=40 ConsoMAX=50 Enclosed=asterix[200-249]
Entity=chassis6 Type=Chassis ConsoMIN=10 ConsoMED=40 ConsoMAX=50 Enclosed=asterix[250-299]
Entity=chassis7 Type=Chassis ConsoMIN=10 ConsoMED=40 ConsoMAX=50 Enclosed=asterix[300-349]
Entity=chassis8 Type=Chassis ConsoMIN=10 ConsoMED=40 ConsoMAX=50 Enclosed=asterix[350-399]

Entity=asterix[0-399] Type=Node ConsoMIN=10 ConsoMED=80 ConsoMAX=400 Enclosed=EnergySocket[0-1],Motherboard,HardDisk,NetworkCard,Memory

Entity=Motherboard Type=Motherboard ConsoMIN=0 ConsoMED=20 ConsoMAX=50
Entity=HardDisk    Type=HardDisk    ConsoMIN=0 ConsoMED=3 ConsoMAX=5
Entity=NetworkCard Type=NetworkCard ConsoMIN=0 ConsoMED=4 ConsoMAX=7
Entity=Memory      Type=Memory      ConsoMIN=0 ConsoMED=10 ConsoMAX=40

Entity=EnergySocket0 Type=Socket ConsoMIN=0 ConsoMED=30 ConsoMAX=60 Enclosed=EnergyCoreXeon[0-5]
```

```
Entity=EnergySocket1 Type=Socket ConsoMIN=0 ConsoMED=30 ConsoMAX=60 Enclosed=EnergyCoreXeon[6-11]

Entity=EnergyCoreXeon[0-11] Type=Core ConsoMIN=0 ConsoMED=5 ConsoMAX=10
```

Only fictive configurations are given since other configuration files can be really huge and fictive ones are comprehensible and possible in the reallity approaching mini clusters.

B Dump

It may be wanted to see what the generated entities and constructed layouts look like from the previous configuration files. Here you find a dump of combined previous configuration files. There is two sections in the dump, first: entities, then layouts. Entities show their data section. Layouts show a browsing of trees going child first then left to right. Each tabulation shows a level of deep.

The file has been shortened due to being huge, it still show the principle without parts too similar.

```
-- entity Bruyeres --
[...]
-- entity Room1 --
[...]
-- entity Row1 --
[...]
-- entity Rack1 --
type: Rack
node count: 2
ptr: (nil)
data racking.coordsy (type: 4): 11
data energy.consomin (type: 4): 1001
data energy.consomed (type: 4): 1101
data energy.consomax (type: 4): 1501
[...]
-- entity chassis8 --
type: Chassis
node count: 2
ptr: (nil)
data racking.coordsy (type: 4): 21
data energy.consomin (type: 4): 101
data energy.consomed (type: 4): 401
data energy.consomax (type: 4): 501
-- entity asterix0 --
type: Node
node count: 2
ptr: (nil)
data racking.coordsz (type: 4): 11
data energy.consomin (type: 4): 101
data energy.consomed (type: 4): 801
data energy.consomax (type: 4): 4001
[...]
-- entity switch1 --
type: Switch
node count: 1
ptr: (nil)
data energy.consomin (type: 4): 31
data energy.consomed (type: 4): 151
data energy.consomax (type: 4): 451
[...]
-- entity rackdoor1 --
type: RackDoor
node count: 1
ptr: (nil)
data energy.consomin (type: 4): 401
data energy.consomed (type: 4): 501
data energy.consomax (type: 4): 601
[...]
-- entity Motherboard --
[...]
-- entity HardDisk --
[...]
-- entity NetworkCard --
[...]
-- entity Memory --
[...]
```

```
-- entity EnergySocket0 --
[...]
-- entity EnergyCoreXeon0 --
type: Core
node count: 1
ptr: (nil)
data energy.consomin (type: 4): 0l
data energy.consomed (type: 4): 5l
data energy.consomax (type: 4): 10l
[...]
-- layout default --
type: racking
priority: 10
struct_type: 1
relational ptr: 0x24f1c98
struct_type(string): tree, count: 416
entities list:
 Bruyeres
 Room1
  Row1
   Rack1
    chassis1
     asterix0
     [...]
     asterix49
    chassis2
     asterix50
     [...]
     asterix99
   Rack2
    chassis3
     asterix100
     [...]
     asterix149
    chassis4
     asterix150
     [...]
     asterix199
  Row2
   [...]
    asterix399
-- layout default --
type: energy
priority: 10
struct_type: 1
relational ptr: 0x2505808
struct_type(string): tree, count: 2836
entities list:
 Bruyeres
 Room1
  Row1
   Rack1
    chassis1
     asterix0
      EnergySocket0
       EnergyCoreXeon0
       [...]
       EnergyCoreXeon5
      EnergySocket1
       EnergyCoreXeon6
       [...]
       EnergyCoreXeon11
      Motherboard
      HardDisk
      NetworkCard
      Memory
     asterix1
      EnergySocket0
      EnergySocket1
      Motherboard
      HardDisk
      NetworkCard
      Memory
     [...]
    switch1
    rackdoor1
   Rack2
    [...]
```

References

[1] 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 *5th Int. Symposium on Cluster Computing and the Grid*, pages 776–783, Cardiff, UK, 2005. IEEE. 8

[2] François Chevallier. Optimisation de la consomation énergétique d'un super-calculateur. 2012. 11, 12

[3] Yiannis Georgiou. *Contributions for Resource and Job Management in High Performance Computing*. PhD thesis, Joseph Fourier University, Grenoble, 2010. 9

[4] GitHub. Repositories: Build software better, together. http://github.com/. 6

[5] Grid5000. Experimental grid platform. http://grid5000.fr/. 8

[6] HTCondor. High-throughput computing software framework for coarse-grained distributed parallelization of computationally intensive tasks. http://www.cs.wise.edu/condor/. 8

[7] LLNL. Slurm: Resource and job management system. https://computing.llnl.gov/linux/slurm/. 9

[8] OAR. Resource and job management system. http://oar.imag.fr/. 8

[9] Sun Oracle. Oracle-sun grid engine manual. http://wikis.sun.com/display/gridengine62u6/Home/. 8

[10] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005. 8

[11] TOP500. supercomputer sites. http://www.top500.org/. 6

[12] Torque. Resource manager. http://www.clusterresources.com/products/torque/docs/. 8

[13] Andy B. Yoo, Morris A. Jette, and Mark Grondona. SLURM: Simple Linux utility for resource management. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer Verlag, 2003. Lect. Notes Comput. Sci. vol. 2862. 9

Abstract

Title:	**Design of a generic framework for the *Slurm* RJMS**
Theme:	Miscellaneous
Keywords:	Industrial Computing, Product development, Maintenance, Eco-conception, System of information

To face an increasing requirement in computing power, supercomputers aggregate more and more computing units. This power requires a large amount of energy that have real consequences on the running cost and on the environment. Moreover communication infrastructure complexity is increased. It is necessary to limit and optimize its use.

The objective of my last year internship as a student of the engineering school **ISTIA** is to study the possibilities offered at the software level in order to optimize the energy consumption and possibly other aspects of super-computers. It focuses on the enhancement of the *Slurm* software utility in that field by the introduction of a new framework proposition.

Slurm is a resource manager. This tool is at the core of the distribution of the calculating tasks on super-computers. It is currently sold as part of the *Bull* HPC all integrated solutions.

This report presents the work completed during this internship realized between may and august 2013 at *Bull* in Ter@tec - Bruyères-le-Châtel, France.

Titre:	**Design of a generic framework for the *Slurm* RJMS**
Thème:	Divers
Mots-clés:	Informatique industrielle, Dévelopement sur produits, Maintenance, Éco-conception, Système d'information

Pour faire face à une demande croissante en terme de puissance de calcul, les super-ordinateurs agrègent de plus en plus d'unités de calcul. Cette puissance requiert une grosse quantité d'énergie ce qui a une réelle conséquence sur le coût de fonctionnement et sur l'environnement. De plus la complexité de l'infrastructure de communication est augmentée. C'est nécessaire de limiter et optimiser son usage.

L'objectif de mon stage de dernière année, en tant qu'étudiant à l'école d'ingénieurs l'**ISTIA**, est d'étudier les possibilités offertes au niveau logiciel afin d'optimiser la consommation d'énergie d'un supercalculateur et potentiellement d'autres aspets. Dans cet optique, il se concentre sur l'amélioration du la solution logistique *Slurm* par l'introduction d'une proposition d'un nouveau cadre de travail.

Slurm est un gestionnaire de ressources. Cet outil est au centre de la distribution des tâches de calculs sur les super-ordinateurs. C'est actuellement l'une des parties dans les solutions de calculs haute performance (HPC) tout intégré vendues par la société *Bull*.

Ce rapport présente le travail effectué durant le stage réalisé entre mai et août 2013 à *Bull* à Ter@tec - Bruyères-le-Châtel, en France.