



*Centro de Investigación en Computación
Instituto Politécnico Nacional*

**Implementation of a self-configuring solution for
job load balancing on a HPC cluster through
virtualization**

by:

Ismael Farfán Estrada

Advisors:

René Luna García

Kenneth Yoshimoto

THESIS

for the degree of

Master in Computing Sciences

(Maestro en Ciencias de la Computación)

June 2012

*Center of Computing Investigation
National Polytechnic Institute*



INSTITUTO POLITÉCNICO NACIONAL

SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

ACTA DE REVISIÓN DE TESIS

En la Ciudad de México, D.F. siendo las 12:00 horas del día 24 del mes de Abril de 2012 se reunieron los miembros de la Comisión Revisora de la Tesis, designada por el Colegio de Profesores de Estudios de Posgrado e Investigación del:

Centro de Investigación en Computación

para examinar la tesis titulada:

**"Implementation of a self-configuring solution for job load balancing
on a HPC cluster through virtualization"**

Presentada por el alumno:

FARFÁN

Apellido paterno

ESTRADA

Apellido materno

ISMAEL

Nombre(s)

Con registro:

A	1	0	0	3	1	2
---	---	---	---	---	---	---

aspirante de: **MAESTRÍA EN CIENCIAS DE LA COMPUTACIÓN**

Después de intercambiar opiniones los miembros de la Comisión manifestaron **APROBAR LA TESIS**, en virtud de que satisface los requisitos señalados por las disposiciones reglamentarias vigentes.

LA COMISIÓN REVISORA

Directores de Tesis

Dr. René Luna García

Dr. Yoshimoto Kenneth

Dr. Gilberto Lorenzo Martínez Luna

Dr. Marco Antonio Ramírez Salinas

Dr. Jesús Alberto Martínez Castro

Dr. Rolando Menchaca Méndez

PRESIDENTE DEL COLEGIO DE PROFESORES



INSTITUTO POLITÉCNICO NACIONAL
CENTRO DE INVESTIGACIÓN

Dr. Luis Alfonso Villa Vargas
DIRECCIÓN DE COMPUTACIÓN

DIRECCIÓN



INSTITUTO POLITÉCNICO NACIONAL
SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

CARTA CESIÓN DE DERECHOS

En la Ciudad de México D.F. el día 4 del mes Junio del año 2012, el (la) que suscribe Farfán Estrada Ismael alumno (a) del Programa de Maestría en Ciencias con número de registro A100312, adscrito a Centro de Investigación en Computación, manifiesta que es autor (a) intelectual del presente trabajo de Tesis bajo la dirección de dr. Luna García René y cede los derechos del trabajo intitulado A solution for HPC cluster virtualization, al Instituto Politécnico Nacional para su difusión, con fines académicos y de investigación.

Los usuarios de la información no deben reproducir el contenido textual, gráficas o datos del trabajo sin el permiso expreso del autor y/o director del trabajo. Este puede ser obtenido escribiendo a la siguiente dirección ifarfan@0900@ipn.mx. Si el permiso se otorga, el usuario deberá dar el agradecimiento correspondiente y citar la fuente del mismo.

Farfán Estrada Ismael

Nombre y firma

Contents

1	Introduction	7
1.1	History	7
1.2	Architectures and uses	9
1.3	Virtualization	14
1.4	Problem approach	17
1.5	Justification	17
1.6	Objectives	17
2	Problem	19
2.1	Scheduling	19
2.2	Motivation, cluster usage	23
2.3	The packaging problem	26
2.4	Job starvation and expansion factor	28
3	Solution Proposal	31
3.1	A cluster that grows and shrinks dynamically	31
3.2	Formal description	32
3.2.1	Good cluster usage with FCFS	33
3.3	General solution	34
3.3.1	Ad-hoc VMs by per node requirements	35
3.4	Software review	36
3.4.1	Virtualization technology	36
3.4.2	Network	37
3.4.3	Cloud technology	38
3.4.4	Resource manager	39
4	Results	41
4.1	Hardware and software specifications	41
4.2	Architecture of Thiao	43
4.2.1	Basic VM management	43
4.2.2	Load balancing	46
4.2.3	Flexible scheduling	48
4.2.4	Advanced management	48
4.3	Experiments	50

4.3.1	Powers saving mode	51
4.3.2	VM on per job basis	53
4.3.3	Performance	54
5	Conclusions and future work	59
5.1	Virtualization and resource managers	59
5.2	Future work	61
A	Middleware	63
A.1	Definition	63
A.2	Classification	64
A.3	Middleware parar clusters	66
A.3.1	Randomly available resources	66
A.3.2	The problem with the access	66
A.3.3	User's shared directory	67
A.3.4	Add and remove resources	68
A.3.5	Statistics	68
B	SLURM	69
B.1	Job execution	69
B.2	Queuing process	72
B.3	Job scheduling	74
C	Scheduling and planning	75
C.1	Planification levels	75
C.2	Kinds of cluster schedulers	77
C.3	Scheduling algorithms	78
D	Graphs	81
E	Source code	87

Abstract

In a HPC (High Performance Computing) cluster the jobs are sent to be executed directly in the physical nodes mainly due to performance concerns, nevertheless such jobs not necessarily use all the allocated resources for many and different reasons; not only that but also in many occasions there are nodes that are not assigned processes to run even if there are jobs in the waiting queue because many of this waiting jobs require more resources that the ones available at that particular moment in time and have to wait for some to be freed.

In this work is presented a way to dynamically create virtual nodes that make use of the extra resources in the nodes in such a way that more jobs can be ran at the same time while sharing resources among them and making use of all the resources that otherwise would be wasted.

Virtualizing an HPC cluster is not a new idea, nevertheless at this time there is no software available that automate the start and shutdown of virtual nodes; here is presented a real implementation and design of a solution that allows us to easily create a virtual cluster, this was made making use of popular software for both cluster and cloud developing an intermediate layer that make them work together.

Resumen

En un cluster de cómputo para HPC (Cómputo de Alto Desempeño) los trabajos son enviados para ser ejecutados directamente en los nodos físicos principalmente por motivos de desempeño, sin embargo estos trabajos no necesariamente utilizan todos los recursos de los nodos asignados por muchos y diversos motivos; lo que es más en ocasiones hay nodos que no son asignados procesos para ejecutar aún si hay trabajos en la cola de espera dado que muchos de estos trabajos en espera requieren más recursos de los que hay disponibles en ese momento en particular y tienen que esperar a que se liberen.

En este trabajo es presentada una manera de dinámicamente crear nodos virtuales que hagan uso de los recursos sobrantes en los nodos de tal manera que se puedan ejecutar más trabajos al mismo tiempo que comparten los recursos entre si y a su vez utilizan aquellos recursos que de otra manera estarían siendo desperdiciados.

Virtualizar un cluster HPC no es una idea nueva, sin embargo actualmente no hay disponible algún software que automatice el inicio y apagado de los nodos virtuales; aquí es presentada una implementación y diseño real de una solución que nos permite crear fácilmente un cluster virtual, esto fue realizado utilizando software popular tanto para cluster como para nube desarrollando una capa intermedia que les permite trabajar en conjunto.

Chapter 1

Introduction

Computing clusters have had a good popularity over the last two decades due to their convenience on helping scientists to solve hard computational problems that require a lot of processing power, with this increasing popularity also comes a series of problems including but not limited to executing the jobs in the arrival order or maintain the nodes of the cluster as busy as possible all the time.

It is important for the reader to understand what are the clusters and why we need them, in chapter 1 we start by reviewing a little about the history of the computers and how it turned out to be necessary to build clusters in order to add the power of many of them to solve a single problem; we'll also review some of the popular architectures and configurations.

With the variety of processors and (specialized) operating systems that were created in the earliest stages of computing, some people had the necessity of running old programs in newer and faster hardware which were not compatible, since it is an important aspect of the present work, also in chapter 1 we give an introduction to virtualization, what is it and why we need it.

The rest of this work is as follows: in chapter 2 we talk about the problem of job scheduling, packaging, cluster usage and job starvation, how they correlate and why it's been so difficult to treat them; in chapter 3 is explained the way we suggest that the use of Virtual Machines (VMs) can be beneficial as an alternative to treat some of those problems of the computing clusters using some known software packages; in chapter 4 the actual implementation of the proposal is described used as an example our testing environment; lastly in chapter 5 a final review of the proposal and results is given together with some unsolved problems and future work.

1.1 History

During the design and birth of the modern computer in 1940, the possibility of parallelism was considered but rejected by Von Neumann, surely there were reasons for that. This architecture now is known as the *the architecture of*

Von Neumann and it has one intrinsic performance limitation called *the Von Neumann bottleneck* which is basically the fact that the processor has to wait while the data it needs to work is fetched from the main memory.

Nevertheless during that same period of time many attempts to create a parallel computer were made, one of the which was a project called ILLIAC IV that is a computer for massive parallel processing. Regrettably there were very few publications about it and it wasn't very successful in the market, so it was abandoned and the area of a parallel computer remained only as research topic in research centers and big universities.

The scientists wanted to verify or disprove things, the engineers wanted to simulate more parts of the hardware of a vehicle, probably simulate the effects of a new drug, the sellers wanted to check their sales or offers as fast as possible. In general, the speed at which a calculation was made could (and can) be the difference on who publishes something new.

Between the 70s and 80s it was realized that microprocessors had a good ratio of cost versus performance compared with big computers, the problem was that they were not very fast individually. The decision made about it was that if it was possible to put many of these microprocessors in a single place together it would be possible for them to have good processing power for a relatively low cost.

The computing clusters are born from this agglomeration of processors or computers connected in a way such that they can work simultaneously in order to solve a single problem or execute a single common program.

All through history different types of clusters and configurations have been invented depending on the specific necessities of the users and clients, and also the available equipment acquisition budget; some examples of the different kinds are given in section 1.2.

So that was the start of the modern computing clusters through history: somehow make many computers work as if they were one in order to solve a single computing problem. However let's remember that was a time of continuous changes around the world of computing, so many different chips were engineered, many different operating systems developed and so many different applications of all kinds.

All those continuous changes and improvements created a completely independent problem: how can we keep on using our software with new hardware if they are not compatible?

It turns out that though software could not always natively be run in newer operating systems or hardware, it was possible to emulate either or both of them in order to run such software, surely it imposed certain overhead, but since newer hardware was very likely to be faster than the emulated one, it didn't really matter much.

This emulation of hardware, software or both evolved in many different ways in order to run all that outdated software, but also could be used as test bed or prototype for new stuff or to study and improve existing one; the branch that is of interest for this document is called virtualization which basically consists on tricking the operating system into thinking that it is running on a hardware

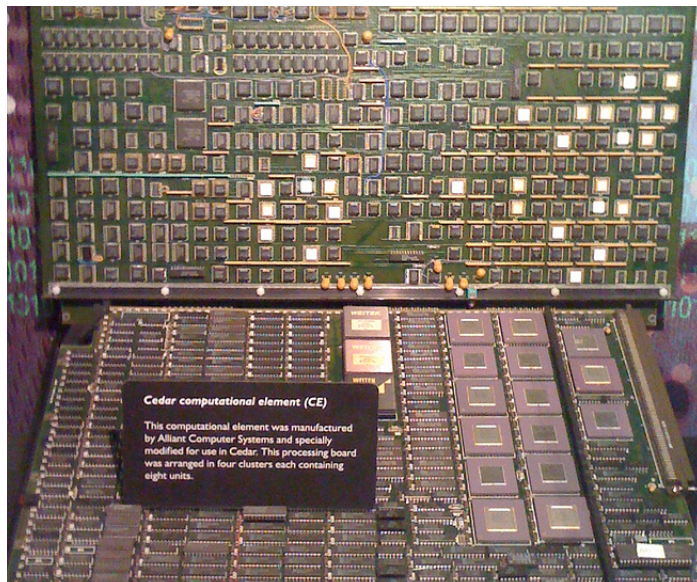


Figure 1.1: A mother board with some processors at the right side and some memory slots.

of its own, we will study more about virtualization in section 1.3.

1.2 Architectures and uses

From the hardware point of view we have many different cluster configurations that goes from putting many boards with their processors somehow together to buying a single big machine with everything integrated, here we give an overview of such configurations or architectures including the current ones.

On one side we have what is called Symmetric MultiProcessing or SMP, it basically involves a single board with lots of slots for (usually) identical CPUs that share the same main memory and are administered by a single instance of an operating system. This architecture had the advantage that companies would by lots of CPUs at a single time, getting them cheaper. One of the first machines known is the Burroughs D825 in 1962. We can see a picture of an SMP board in figure 1.1.

It was not always possible for the companies or laboratories to buy a whole lot of identical processors, and they not always liked the idea of trashing all the costly hardware they already got in order to buy new one. The solution for this was the Asymmetric MultiProcessing architecture or AMP, it was the same basic idea of the SMP, but this time it was possible to use different kinds of processors (or at least different versions or revisions of a particular brand). However wen this idea surged between the years 60s and 70s, the operating systems were not

prepared to run in many processors; so in this case the operating system ran in the main processor and when a job entered it sent it to a particular processor to be executed.

One supposed enhancement to the SPM architecture was something called Non Uniform Memory Access or NUMA; the main difference between them is that the NUMA architecture uses a dedicated memory for each processor at a place near of it. However it also introduced the problem of data access, a protocol is needed in order to get or modify the data from the memory of other processor to ensure data integrity and make sure that every processor is working with the latest version of the data.

This kind of equipment could and can be bought from hardware companies specialized in High Performance Computing like Cray Inc. Sometimes they come with their own operating system, tools and connections.

The problem of data integrity is not specific of the NUMA architecture, almost every parallel program will have to make sure that every thread is working with the latest available copy of the data; NUMA happened to work with this problem at a hardware level, some of this was latter adopted by others, the main subject about it is the cache coherency [38].

The SMP architecture has got very cheap and thereof popular in the last few years with the introduction of the multicore processors, each core has everything it needs to execute a program and the only thing that is shared among them is the access to the memory; at the time of writing it was possible (and affordable) to get a desktop computer with a 8-core processor.

Continuing with the SMP architecture, before the 8-core processors came out there was another way to make your own SMP or AMP cluster using lots of independent computers connected to a local network through high speed switches, the users could connect to a login server and send their jobs for execution from there.

So far we have described clusters from the hardware point of view taking emphasis on “user jobs”, however there are some more specialized kind of clusters that are built to run a single application or a small number of them, these are typically service or application clusters with fail-over capabilities through redundancy.

An example of this a software cluster is in the case of a database; sometimes databases handle too much information and are asked to execute hard queries on it, this can be a heavy task if many of this are requested every time, so a cluster comes in handy, create a master database server and then replicate the data across the slave machines, then distribute the queries requests among all of them to ensure responsiveness. A particular example of this kind of software is MySQL-cluster.

Another typical scenario for this clusters are the web servers; back in the days of HTML 1 this servers only sent the page that was requested, however lately with the introduction of software like PHP, ASP, etc. their task has increased considerably since now the servers have to compile the page before actually sending it, this can be a problem with big and heavy pages, so sometimes the different processing bits of the server are distributed across the cluster, so one



Figure 1.2: A shelf with random computers that for a Beowulf cluster.

machine does the SQL queries, other compile the page, etc.

More recently, with the introduction of cloud computing, virtualization clusters have surged, basically you have an image of a operating system with the services and libraries required, if one image is not responding because there are many requests, just launch another one and make it execute some of the requests, when the load decreases shutdown one of them; we will talk more about virtualization in section 1.3 and how to use it for High Performance Computing (HPC) work loads in chapter 3.

After all this talk about clusters it's probably time to talk about how to build your own HPC cluster, since there are lots of ways, we'll only cover some of the popular or interesting ones, from using your coworker's computers to buying new dedicated hardware. For a more extensive explanation covering from the electrical installation see [7].

Probably the most popular configuration of the "build your own cluster" kind is Beowulf, the most basic architecture consists on taking any computer you can get for exclusive use of the cluster, connect them through an Ethernet switch, decide which will be the master node and you are done; it's recommended to pile them somewhere to save space like in the figure 1.2, we will talk a little bit of the software used latter.

The advantage of the Beowulf cluster is that the machines are dedicated for its use, so the administrator have a complete and centralized control of them; when a process is executed in one node, it will give all of its resources to the process allowing it to run at full speed every time.

The disadvantage is that if there are no jobs that can be run by a particular node at some point in time, the node will be there waisting energy doing absolutely nothing; with the current costs of electricity that is an undesirable thing,

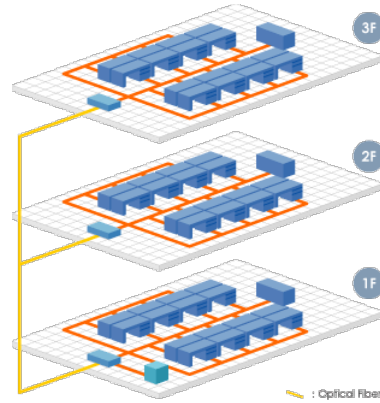


Figure 1.3: A COW cluster using all the workstations from 3 different floors.

we want the nodes of the cluster to be busy as much as possible or there is no point in having them at all.

There is another very interesting possibility, let's say that everyone in the office has assigned a computer, like in figure 1.3, the coworkers or users most likely won't be using even half the processing power of their respective machines on common administrative or office tasks; it is possible to detect which computers are idle and which are not and select them to execute jobs, this is known as a Cluster Of Workstations (COW).

The COW architecture comes in two main variants, it can be specified from the beginning how much of the resources of each node can be used for the cluster at a given time, or it can be dynamically detected depending on the load of the machine and other factors. The disadvantage of this architecture is that it is possible that the jobs are affected by the work being done by the user at random points in time, or the jobs affect the user.

For this project we are using a mixture of the ideas surrounding the Beowulf and COW cluster architectures, having a bunch of dedicated nodes, but monitoring them frequently to know how busy is each one and basing the job assignment on the load. We will talk more about this in chapter 3.

As we can see in figure 1.2, an average Beowulf cluster takes a lot of space which limits badly the amount of computers that can fit into a room, one solution for this is to trash the cabinet and keep only the boards with the processors; depending on the configuration a hard drive may not be necessary either, allowing to save even more space. However, professional production systems will use a tower like the one shown in figure 1.4, this kind of tower in particular is around two meters tall and can hold around 60 computers, these are specially designed to keep the machines as cold as possible.

So how do you configure the cluster to make the machines execute jobs automatically anyway? As usually with technology, the answer is "it depends on your needs" however here we'll show one of the most common solutions, what



Figure 1.4: A rack for cluster with capacity for 60 mother boards.

you need is a shared file system to be able to find the files regardless of the node that execute the job; an authentication system to allow the users to login securely in the system; and a resource manager that will receive the requests and distribute them across the available machines; it is also recommended to have the same operating system in all the nodes, unless you know what you are doing.

While most modern operating systems (OS) can be used for building your own cluster, the GNU/Linux family is the preferred one for various reasons, including the fact that most distributions are free, and there are others that have commercial support like RHEL¹; at the time of writing, 91.4% of the clusters in the top500.org used a GNU/Linux OS. Another good reason for using GNU/Linux is because it comes with most of the things needed to build a cluster out-of-the-box.

While most big clusters will look like the one in figure 1.4, an easier to understand topology will look a lot like the one in figure 1.5, there we can distinguish processing nodes with different characteristics at the left that will be the ones executing the jobs, one master node or login server in the middle that serves also as gateway for the nodes that need access to the internet, and finally some file servers that will hold the data of the users. Finally we can see that all the nodes are connected through a high speed switch.

A shared file system (FS) is important because the users needs to be able to get the files that are required to execute the programs from any node, since nodes are assigned quite randomly, and some times the users are not allowed to access the nodes on their own, a centralized place to store everything is

¹Red Hat Enterprise Linux

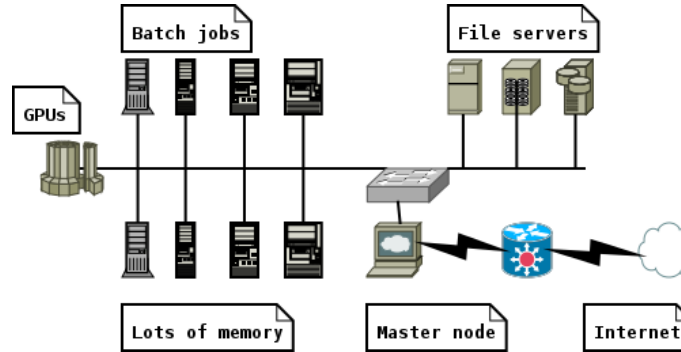


Figure 1.5: An example of a Beowulf cluster to distinguish between processing nodes at the left, a login server in the middle and some file server at the right.

desirable. One popular solution is the Network File System (NFS) that comes by default with most GNU/Linux OSes.

The authentication system is important because is desirable to make sure that the person using the system is the one that was given access to it; also it may (or not) be desirable to treat all users equal, specially if some are paying to use the infrastructure; finally it's not a good idea to allow the users to access directly to the nodes because they may interfere with the jobs of other users, however this is a more advanced configuration. GNU/Linux clusters will allow the access through Secure SHell (SSH) that allows to encrypt the communications, is very configurable and comes with every GNU/Linux system.

There is one particular piece of software that makes it possible to efficiently share the computational resources among all the users of the cluster, that is the middleware, or more precisely the Resource Manager (RM) which job is to monitor the existing resources and share them among the users; its main function can be explained more easily from the figure 1.6: the user will send (or request) a job execution to the RM, the RM will put the job in its queue and allow them to run according to the scheduler rules used and the amount of resources available; finally the job will run in a set of assigned nodes until it finishes or its time runs out. We will talk more about the middlewares in appendix A.

1.3 Virtualization

We have talked only a little bit about virtualization so far but haven't said how it works or why is it important, so in this section we are giving a more in dept explanation of what is virtualization, what it does, how it works and some more information that may be interesting for those that are working on the area and also for those that are not familiar with it.

Resource virtualization is a joint technique of software and hardware that

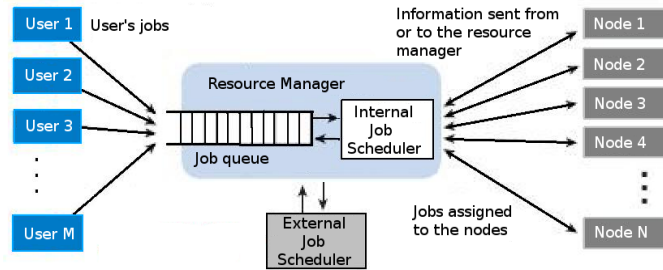


Figure 1.6: Life of a job in a cluster: enter the queue, wait for resources, run until it finishes or its time runs out.

allow us to trick a process into thinking that it is being executed inside a physical node while that's not actually the case, so we can execute many different applications inside one computer and at the same time isolate all of them.

Virtualization is also a very broad subject, thereof in this section we give only the basic concepts that helps to understand how it works and how to apply it. We start with a very general definition of what virtualization is [43] “Virtualization is a technology that combines or divides computing resources to present one or many operating environments using methodologies like hardware and software partitioning or aggregation, partial or complete machine simulation, emulation, time-sharing, and others.”

Partitioning From a multitude of equal resources a piece of one is shared.

For example, in the case that we have two network cards but we want to present only one.

Aggregation The opposite of partitioning, we want to present more resources than the ones actually available. For example we can present more RAM memory through the use of a swap partition.

Simulation Through software the instructions of the program are interpreted in order to reproduce the behaviour.

Emulation Its a way to model the device for which the application was written in order to allow it to execute.

Both, simulation and emulation require support through software in order to run applications, this make the programs execute slower than in the original hardware, this is one of the reasons for which neither of the two were adopted for use in high performance computing where is used all the computing that is possible to get.

The virtualization technologies have improved a lot in resent years, now the processors provide hardware extensions that make the emulation easier, for example AMD-v from AMD and VT-x from Intel are hardware extensions that are used by products like VirtualBox, VMWare and others to make the virtual

machines a lot faster; nevertheless they still need some pieces of software that modify part of the guest OS so that some devices work correctly, for example it's usually needed to install some device drivers provided by the virtualization vendor.

In grid environments many users donate their dead processor cycles² for use in the grid that they are members of, this potentially gives access to their (personal) machine to hundreds of users of the network, which is not a good thing from the point of view of security; for this reason since some years virtual machines have been used for the execution of scientific code in grids, the user only needs to give access to the virtual machine and in this way the programs run in a completely isolated environment leaving alone the files of the user.

Even though the hardware assisted virtualization provides a good performance, the fact of having to install and upgrade specialized device drivers on the guest OS and the latency for input and output devices were a hindrance for its use in high performance computing, so much that at the time of writing of this document only very few serious publications about its use for HPC were found [20, 40].

The fact that there are not many publications about the use of virtual machines for high performance computing does not mean that there is no interest in it, in fact, the thing is that currently there is no important supercomputing center that uses them as part of their infrastructure, nevertheless is an alternative that is being explored [50].

Since many years ago the concept of “paravirtualization” was conceived, though this technique became popular recently with the introduction of the Xen hypervisor³ in 2003 [3]. This technique consists in allowing most of the instructions to execute directly in the bare hardware, without relying on a software that interprets the application, this allows the programs to execute roughly at the speed that they would execute if they were in their own physical machine.

Besides Xen, there is another popular hypervisor called KVM (Kernel Virtual Machine), that like the former, provides a good performance of the virtual machines and is used a lot lately, among the reasons for this popularity is the fact that is free and comes included as part of the Linux kernel since version 2.6.20, this makes it very easy to configure.

With this and other paravirtualization alternatives a new interest on using VMs on HPC surged; numerous studies of performance have been made regarding the use VMs for HPC for example in the area of memory access [26], also for reducing the input and output penalty [37], and methods for optimizing scientific libraries like MPI [25].

The main problem with solutions like [50] is that it only propose the use of virtual machines to build and administer a supercomputing cluster and gives much of the theory about job scheduling and resource administration, but it doesn't provide an actual way of building or configuring a virtual cluster.

This little gap is taken into account by various cloud technologies which

²Time that the computer is on but is doing no work.

³Or Virtual Machine Manager (VMM)

provide means for realizing much of the administrative tasks that get us closer to a virtual cluster, for example the EC2 from Amazon and the One-Click Cluster from Nimbus; the problem with both of this is that they are independent solutions and doesn't integrate seamlessly with the existing infrastructure.

We talk a little more about the building a virtual cluster in chapter 3 since that is an important piece of information for this work.

1.4 Problem approach

Nowadays the use of computing clusters to run scientific applications is a widespread trend in many different social communities due to the huge amount of resources that are easy to request; however this massive amount of resources not always is used at its fullest for many reasons including the fact that packaging all the applications given their expected runtime and resources is a very complex problem. On the other hand the virtualization of the Operating Systems has been getting popular specially for server environments because it allows to easily share computing resources across isolated applications or (virtual) servers and thereof increase the utilization of resources of the hosting nodes (ie: hardware). On one hand we have a computing cluster whose resources are not used at its full potential and on the other hand we have a technique that allows us to easily use more of the resources of the nodes; from here the idea of using virtual machines to create a computing cluster for scientific applications was born; we wish to be able to monitor the nodes of the cluster in search of available resources and allocate them for the use of the jobs, this way it is believed that it is possible to dispatch more jobs in the same amount of time.

1.5 Justification

The interest in the use of virtualization for HPC workloads is slowly but steadily increasing across the community due to the benefits that it provides and the active development and improvement of such technology, however at the time of writing there is still no free solution available for joining both worlds, instead there are applications that do either of the two. The main point of this thesis is to show both a general architecture and the actual implementation of it for creating a virtual cluster that automatically starts and stops virtual machines based on the amount of jobs (applications) waiting for execution and the amount of resources left unused by the jobs that are executing already.

1.6 Objectives

General objectives:

- Evaluate a general architecture for a virtual HPC cluster

- Develop a software for automating the management of a virtual HPC cluster

Particular objectives:

- Evaluate the possible gains in utilization under the job per node granularity.

Chapter 2

Problem

With the massive collection of computers that form a cluster it is normal that many different problems arise. From the scheduling point of view we have the problem of deciding in which order to send the jobs to the cluster, this in turn affects the average usage of it because sometimes it's necessary to wait for resources, the later is solved through a packaging optimization algorithm, but this might override the desired schedule and move certain jobs again and again preventing them from executing.

In this chapter we give a comprehensive description of this problems, along with some of the proposed solutions for them. To our knowledge there is no way to solve all those problems, at once, or at least not an implementation of one, and here we describe why.

2.1 Scheduling

A schedule is a execution plan for various tasks, this is true for both the personal life of an individual and for the jobs in the computing clusters. The difference could be that one person is restricted to maybe one or two tasks at a time while clusters can allocate a very volatile number us tasks at the same time due to the numerous resources.

Since many years ago, specially between 1900 and 1980, various techniques for scheduling and planning were developed to make an efficient use in Shop Floor Control (SFC) [23] and based in many different criteria, for example those based in job attributes, or in the time required to complete it, the type of jobs in the queue, etc.

It's convenient to describe some of the characteristics of this different philosophies of SFC because some of this were adopted, adapted or ported for the use in computing cluster scheduling where there are also many different kinds of jobs that arrive randomly and from which is necessary to select and optimum way of executing them.

It is possible to think in two possible generic models for planning jobs: a

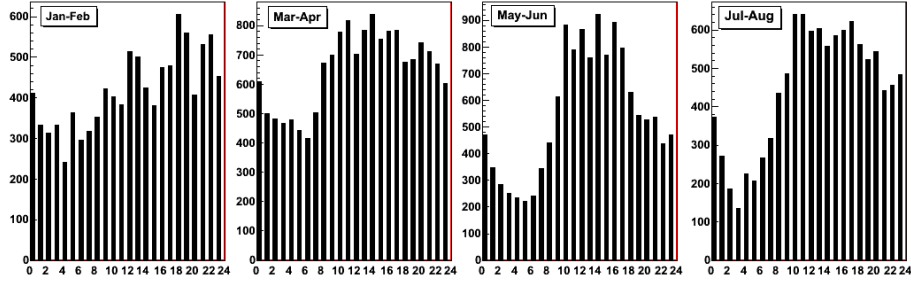


Figure 2.1: Number of jobs that arrived on periods of 2 months in the cluster of ANL during 2009 per hour of the day.

static one in which all the tasks known are planed at once and no new tasks are accepted until all of them are dispatched and the schedule gets empty; and a dynamic one where new jobs keep getting into the list of pending tasks and are scheduled on the go. The static model is very unlikely to have a practical use in practice, so for the current work we think on the dynamic and more flexible model.

The dynamic model for planning is necessary because there are job run requests arriving at the cluster at all times, probably not so many at 3 in the morning, but still some do get into the queue at that time just as we can see in figure 2.1 where we can see an histogram of job arrival per hour of the day in the Blue Gene cluster of the ANL¹ from January to August 2009; it's very similar to the result that we can see in [35], so the scheduling must be flexible; also let's notice that some jobs get canceled before or during their execution leaving holes in the schedule.

How can we select the order in which to queue the jobs anyway? As we can see in [23], by 1989 there were already more than 30 different algorithms to schedule tasks: based on priority, properties, processing, etc. also global and local rules. Each one of this algorithms have their own pros and cons, though we will not study them in detail, let's just see what are they about.

The *priority based* rules make a schedule of the pending jobs in such a way that all the machines get busy, once any machine finishes it selects the next task among the list of pending ones, in order for the machine to make such selection, a new schedule must be computed.

The priority functions *based on properties* assign a priority to the jobs independently of the urgency of realizing other tasks, it only weights the different properties of each individual and use them a measure, there is a variant of this one that also consider other tasks though.

The rules based on *processing time* assign a priority based on the time required to complete the job, for example, it is possible that the longest tasks are considered more critical than shorter tasks, in the other hand, it is possible that long tasks are so rare that they are not really important, so they can actually

¹Argonne National Laboratory.

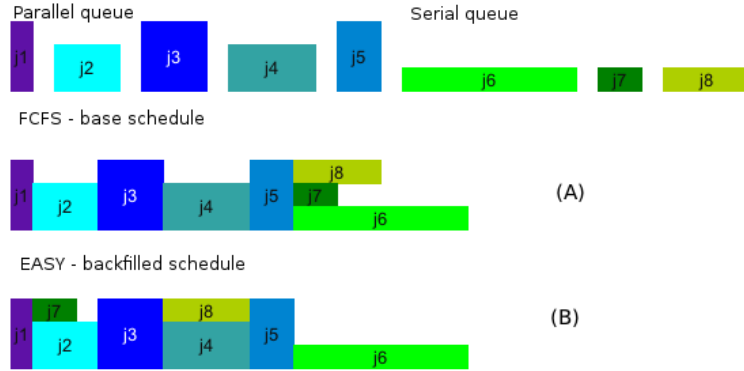


Figure 2.2: An example of the result of scheduling some jobs using a) FCFS and b) EASY backfilling.

wait for the resources to get available.

Finally, in the event that there are various queues that happen to share some resources, we can think of *local* and *global* rules; the local ones just ignore the other queues, while global rules do care about the status of other queues, so for example if there are jobs that can only be executed by certain subset of shared resources and others that can be executed by any, they try to take that into account.

Leaving all those algorithms aside, let's see which are the ones actually used for scheduling clusters: in literature we will usually find FCFS (First Come First Served) and EASY backfilling [24, 33, 41, 46, 52], the former one because is the simplest that one can think of, and the later because it happens to give a fair performance and is easy to implement.

As we can see in figure 2.2a what FCFS does is to try to run the jobs in the same order that they arrived into the queue, if there are not enough resources to run a job it waits for them to free until the next job in the queue is able to run, this has obvious problems since the free resources are being wasted meanwhile. The good thing about FCFS is that there will be no complains about why a job that was sent after another of the same priority ran first.

In 2.2b we can see how EASY backfilling works, it basically will create a schedule just as FCFS, then it will search for holes or wasted resources in the schedule and will try to fit jobs that will execute later into the available holes, this is a good thing because it will keep all the machines busy. This backfilling comes basically in two flavors, one is *conservative* backfilling [42] which creates the whole schedule and then backfills jobs *only* if by doing so it doesn't postpone the scheduled start time of other jobs; the other flavor is *EASY* backfilling [36] which will schedule only the next job in the queue and allow to run other jobs as long as there are enough resources for them to start executing.

Because of its simplicity we can categorize [24] FCFS into the group of *queue based* schedulers since it will always take the next job in line regardless of the

rest of the jobs; while backfilling is more a *planning* based scheduler since it will have an execution plan of some (EASY) or all (conservative) the jobs in the queue and then will try to optimize it.

Another issue caused by this very good backfilling algorithms other than the mere fact that small jobs can keep on skipping the line to execute before others is something called *job starvation* which occurs when a jobs is postponed again and again and doesn't get executed; we talk more about job starvation in section 2.4.

The problem of scheduling is a very important one, it's not only about deciding what goes first than what or how busy are the machines, there are also political implications, for example in [46] we can see many of the decisions around the design of the scheduler *Maui*; things like *quality of service (QoS)* which is the use of privileges or policies based on users, groups, types of jobs, etc. in order to decide how to prioritize the jobs; *fare share* which is making sure that everyone gets resources at some point while preventing a single user from taking all the cluster for themselves (for example in cases when a single user submits hundreds of jobs at once).

Maui is not the only scheduler using all that concepts for planning purposes, and also those are not the only thing to consider that will take a toll in the efficiency of the scheduler's efficiency; schedulers like OSCAR [12], SLURM [30], PBS [18], Catalina [55], and others [54] include considerations about QoS and fare share into their scheduling mechanisms.

There is another problem we haven't mention yet, in order to calculate a schedule, the expected runtime is needed or else there is no other option but to execute the jobs as they arrive; however, what happens when a job doesn't finish in the planned time? there are mainly three options for an event like that:

- Leave the job running and postpone the start time of the waiting jobs that depend upon the finishing of the current jobs. Let's notice that this might render useless the whole schedule as it will move the end time for as much as the extra time required for the job to finish.
- Start the next job anyway; this is very much a poorly monitored solution, two jobs sharing a single node could interfere with one another as the first would be stealing time from the second.
- Kill the offending job and start the next scheduled one in order to keep the current deadlines.

Some companies expend lots of money in products like MOAB in order to have a good scheduler that can provide lots of functionality and good QoS, so the first option is rarely used, expect in old and specialized clusters that doesn't require the user to provide runtime estimates; to our knowledge the second option is not really used expect when there is no resource manager installed in the cluster; finally, killing the job whose time expires seems to be the preferred behavior as its the default in many resource managers including TorquePBS and SLURM.

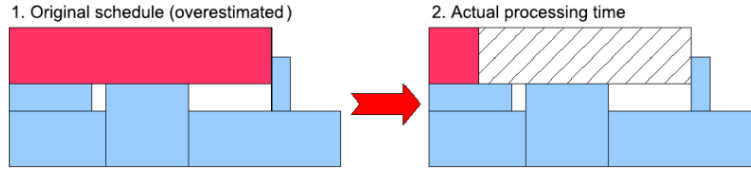


Figure 2.3: 1 the user requests enough time to guarantee that the job finishes. 2 the job actually finishes before the allocated time.

Estimating the runtime of a job is very complicated, specially when it's a parallel job due to Amdahl's law [7,44,47]; it's not only about dividing the time required to run the serial version by the number of machines to use, but it's actually necessary to take into account the fact that there is always a serial part of the program and one or more parallel parts.

Amdahl's law added to the fact that every generation of hardware has improvements or drawbacks in different areas makes calculating the expected runtime a nuisance, and even after this the users of the cluster have to deal with the fact that if the estimation is short the job will be killed make the users overestimate the runtime of their jobs [2,42,51]; so an scenario like the one in figure 2.3 where the job doesn't use half of the requested time is quite common.

Since we don't want to waste all that extra time freed by jobs that finish earlier than expected, and is very unlikely that someone will complain for their jobs finishing sooner, the schedule is recalculated or optimized quite frequently, there's even work being done in schedule optimization [33] and also in predicting the runtime of the jobs before creating the schedule [41,52].

Even though there exists already so many awesome scheduling algorithms based on all kinds of philosophies like profiling the applications [21], predicting the performance [29], using job statistics and monitoring to schedule [34] and others; current cluster middleware keep using the simplest scheduling algorithms specially due to the fact that scheduling thousands of jobs at once can take some time even with today's computers, the algorithms are complex to program and generate a big latency or require lots of memory, so most of them stay in the publications and never get used.

While this section was mostly about the algorithmic approach for the scheduling problem, there are more computational approaches, we show some of them in section 2.3, however even if the approaches are different, the purpose is the same, to increase the utilization of the cluster while trying to keep the user from complaining.

2.2 Motivation, cluster usage

Let's begin by describing what usage means in the context of cluster computers, thought it's similar in other contexts; usage is the measure of how much the resources are being *used*, we can break it in many levels; the higher level we

are concerned about is the *cluster usage* which is how many of the computers of the cluster are working in average over the course of time; of course, this is a continuous function.

We can calculate the usage of a resource at the time t using equation 2.1 where $U(t)$ is the utilization at time t , $r_a(t)$ are the total resources available at t and $r_b(t)$ are the resources busy at t ; so for example, if we have 64 computing nodes and 50 are working then the cluster usage is 84.375% *at that moment*.

$$U(t) = \frac{r_a(t)}{r_b(t)} \times 100 \quad (2.1)$$

Knowing the cluster usage at certain time is not really useful, we are usually concerned about the utilization during a period of time, this could be a continuous function, however the computers updates the usage static in a discrete manner, so we can use equation 2.2 where i is the time of the first measure we have, j is the time of the last one and n is the number of measures.

$$U(i, j) = \frac{\sum_{t=i}^{t=j} U(t)}{n} \quad (2.2)$$

While one of the main objectives of scheduling is to maximize the utilization of the cluster, it is known that the average usage of the nodes rarely goes over 60% [17, 42, 53] (this studies suggest that is between 7% and 22%, but the communication speed has improved a lot since, so probably that's not the case anymore).

Why do we care about utilization anyway? To explain that let's take the (very conservative) example of TCO (Total Cost of Ownership) from [16]; if a 128 node cluster costs \$67,300 USD/year just to run and the average node utilization in one year is 50%, that means that we are waisting \$33,350 USD/year on electricity, now let's say that with the present work we manage to increase the average node utilization from 50% to 60%, that's a 10% increase that accounts for \$6,730 USD/year.

Probably \$6,730 per year doesn't sounds like much, but production systems nowadays are much bigger than that, at the time of writing the machine in rank 498 of the top500 supercomputers was a xSeries x3650M3 with 9084 cores, supposing that each node had two six core processors, then that system had around 757 nodes, so a 10% increase of utilization on that would account for \$39,800; but a kilowatt doesn't costs \$0.10 per hour, and the power consumption of a 12 core node is not the same of a 4 core one, so we can expect the actual gain from a 10% increase to be higher than that.

In the figure 2.4 we can see an example of a schedule, this is not the graph of a function, instead it represents what was each of the nodes doing at a particular moment of time; in the **X** scale we can see the **time in hours since the cluster was turned on**; the **Y** scale represents the worker nodes of the cluster, in this particular case there are **1024 nodes**, for now we don't care about a particular set of nodes and there are too many of them so the scale is not shown; a single job is represented by one or more rectangles of the same color that start and

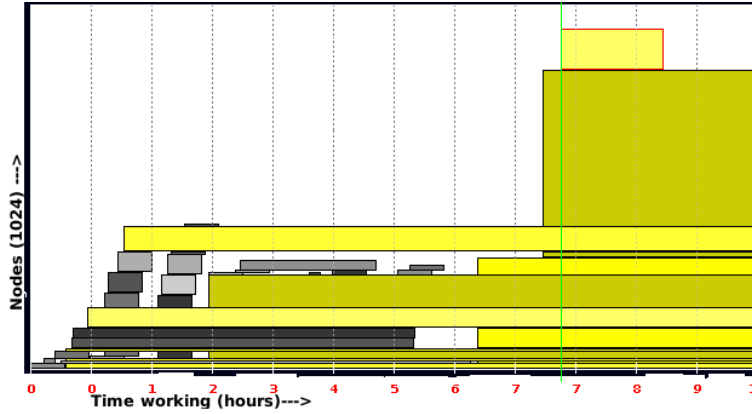


Figure 2.4: The usage of the cluster for around 7 hours is not even 30% because there are no jobs in the queue.

end at the same time, they don't need to be continuous and the decision of how to allocate them is up to the scheduler; the vertical line represents the **moment in time when this snapshot of the schedule was taken** (due to the fact that the schedule changes dynamically over time).

Now that we know why the utilization is an important subject, we can pass onto some of the reasons for the cluster not being used at 100%; one reason is simply because there are no jobs in the queue, and therefore nothing to do as we can see in figure 2.4 where more than half of the cluster got wasted for a few hours. This event can not be helped, no matter how good is the scheduler, if there is nothing to schedule the whole cluster will sit idle.

The second possibility, and the one anyone working on scheduling is concerned with, is when there are jobs waiting to be executed, like in figure 2.5 but none of them can be executed right away because there are not enough resources, so it's necessary to find a way of keep as many of the resources busy as possible at all times; this is also part known as the packaging problem, we explain a little more of it in section 2.3.

There is also another important aspect to know about the schedules shown in figure 2.4 and 2.5, that is not a map of the usage of the cluster at certain point, but actually a map of what machines were given something to do during that time; the truth is that even if the machines were given something to do, they are not necessary using all their resources [53].

Commercial resource managers for computing clusters doesn't monitor the resources of the nodes, they limit themselves to assigning jobs to the nodes; it is known however that duplicating the number of nodes used to solve a problem doesn't cut the time needed to complete it by a half, so this means that the rest is wasted CPU time, time that could be use to run other jobs.

Other kinds of middleware do care about CPU and memory usage of the nodes they are administrating, for example SQL clusters use that information

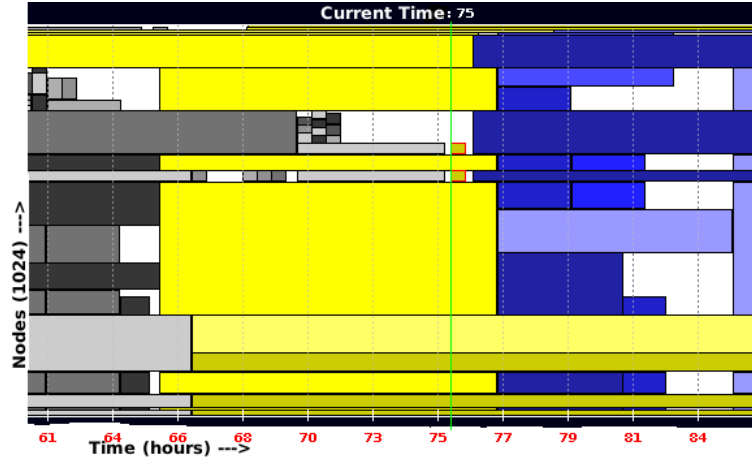


Figure 2.5: There are lots of jobs waiting for execution but non of them fit in the holes, so the cluster gets wasted during that time.

in order to direct the requests to the nodes, however this is more a load balancing strategy since the only thing different is the query, but the data and the server are the same on all nodes. Also the VMM (Virtual Machine Monitor) use to monitor both the hosts (physical nodes) and guests (virtual machines) and use that information to take decisions concerning VM allocation and load balancing across the cluster of VMs.

In chapter 3 we describe our proposal for increasing the cluster usage by using mixing two kinds of middleware, one VMM that is used to start and stop virtual machines depending on the load of the hosts, and one resource manager for computing clusters that allocates jobs on the virtual machines.

To understand better why middlewares seems to do so many different things, which they actually do, refer to appendix A, also for the particular case of the SLURM middleware refer to appendix B.

2.3 The packaging problem

Packaging, when talking about scheduling, is a problem that consists on trying to cover as much of the execution plan area as possible in the smallest amount of time; as we can see in figure 2.5, the scheduler actually did everything it could to try to fill the gaps, but as there were not enough small jobs that could fit in them, some resources still got wasted for some time.

The problem with the scheduling algorithms described in section 2.1 is that they try to make the plan using non-overlapping blocks as width as the number of resources requested and as long as the time required (including the overestimated time), so they heavily depend on small jobs to fill the gaps; big jobs get postponed to start in a future time and are rarely eligible for backfilling while

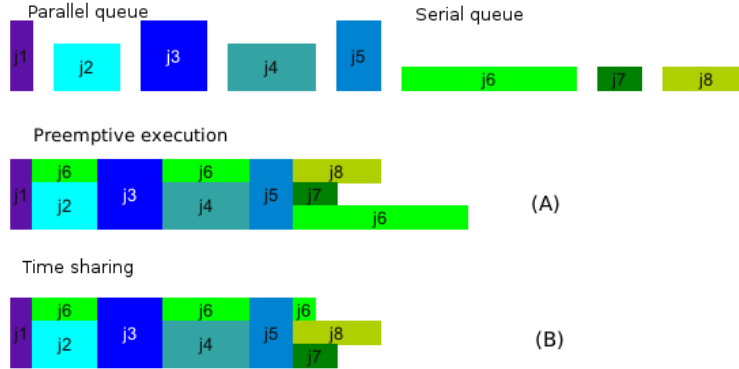


Figure 2.6: Run the jobs as long as there are enough resources. A) If it doesn't finish, kill and re-queue the job. B) If it doesn't finish, suspend and resume again when possible.

small jobs not only skip into the line to get executed before but also are eligible for backfilling.

There are two main ways of doing a better execution plan without depending so much on small jobs that are from the computational point of view: those are *space sharing* and *preemptive execution*. The basic idea behind both techniques is depicted in figure 2.6, both require support from the resource management middleware.

The first of the techniques is the preemptive execution, it allows on allowing the *participant jobs* to be executed as long as there are enough resources available even if there is not as much time as the one requested by the user to run the job; since the time requested is usually greater than the time it actually takes the job to finish, this is a little bit of a gamble because the job may actually finish in time, however, in case it doesn't finish, it is killed to comply with the execution plan of the rest of the jobs and the killed one gets queued again as we can see in figure 2.6a.

Not all the jobs are suitable for preemptive execution because some of them need to do some housekeeping tasks before or after the execution like fetching or (un)compressing files used throughout a simulation or whatever the job is meant to do; this kind of jobs could leave random files spread across the nodes that they started running on, or if they depend on the successful execution of some particular command that destroys the input pile (like `gunzip(1)`) they could just crash the next time.

For some resource managers, this kill and requeue policy is only used in case of a node failure, so the rest of the job is killed and requeued before it crashes because a node died. So this usually has to be enabled from the scheduler itself and also the job needs to be sent to a queue that supports preemptive execution.

The second computational approach for filling the holes in the schedule and package the jobs more tightly is *time sharing*, it not only requires an special

queue and the explicit enabling of it in the configuration of the scheduler, but also requires special support from the operating system and the software and libraries being used.

Time sharing, like preemptive, allows a job to run as long as there are enough resources available even if there is not enough time for it to finish, but unlike the later, when it is time for the scheduled job to begin running, the other job doesn't get killed, instead it is paused by the system and saved to the disk so that when another set of resources of the same size is available it can be resumed on those nodes as we can see in figure 2.6b; the job gets suspended and resumed until it finishes.

Time sharing is even trickier than preemptive execution, it requires the operating system or the libraries or both to support the suspend resume features in order to work. For example, the team of OpenMPI had to implement a particular solution to suspend/resume parallel jobs using it [27], because there would be problems if the job was suspended at any time and there was some group communication or synchronization at the time; also for non MPI jobs, the OS have to have support for such a feature, in the case of the GNU/Linux OS, BLCR (Berkeley Lab's Checkpoint Restart) [15] provides this functionality though a kernel module.

While this solution partially solve the problem of packaging (because it may run-out of participant jobs), it's not always available, for instance, at the time of writing BLCR is not compatible with Linux 2.6.39 and up; also this software solutions are *scheduler based*, so it leaves out the other problem we are to take care of: it ignores the actual utilization of the nodes and only supposes that they are running at full speed.

For more information about scheduling algorithms see appendix C.

2.4 Job starvation and expansion factor

The use of eXpansion Factor (XFactor) and job starvation have been suggested as metric for scheduling and backfilling jobs, and also as metrics for the effectiveness of the scheduler, respectively [48].

The XFactor is basically the ratio of the amount of time that pass since a job entered the queue until it is expected to finish executing taking into account the runtime estimate, the basic way to calculate it is shown in equation 2.3.

$$XFactor = \frac{waittime + estimatedruntime}{estimatedruntime} \quad (2.3)$$

So basically, the greater the time it wait and the shortest the estimated time, the priority of the job goes up; this scheme will give preference (too) to the small jobs that are already skipping the line through backfilling. This also doesn't count the width of the job (ie: the number of nodes requested), so the *weighted* version of the XFactor.

While the XFactor is more a way of prioritizing, the job *starvation* is how well it is doing in terms of starting its execution compared with the the time it

would have started to execute using the conservative backfilling; lets remember that conservative backfilling schedules an start time for every job in the queue and is willing to let the job start before but not after that time.

If using conservative backfilling a job would have started at certain time, and using EASY it starts *after* that time, it is said that the job was starved. Reviewing the results show by [48], we can see that wide jobs tend to be starved, while small and short ones get preference. This is very intuitive, one cannot let a number of jobs skip into the line without others suffering the consequences.

With the current work we want to be able to decrease the overall job starvation while keeping a high real cluster usage and omitting unnecessary priorities for small jobs that end up starving wide jobs.

Chapter 3

Solution Proposal

There have been some studies about the use of Virtual Machines (VMs) for High Performance Computing (HPC), some claim that virtualization is good for HPC because it allows easier migration and testing of new platforms [40]; others claim that the performance penalties are low enough for using in HPC systems [56]; there is who claims that hypervisors still need to be improved [20], others actually provide enhancements [26]. The truth is that virtualization has been improving quite a lot since most of those works were released and no one seems to suggest that VMs shouldn't be used for HPC.

Based on all the previous evidence, and some more [25,37,49], we present an actual architecture for using a virtual cluster for HPC workloads that aims to reduce the overall job starvation and increase the cluster utilization while also allowing to do load balancing across nodes and provides a more flexible schedule.

In this chapter we describe the theoretical part of the solution along with some virtualization, cloud and resource managing software we evaluated in order to do the proposal. In chapter 4 we present the software written to implement our solution along with some experiments.

3.1 A cluster that grows and shrinks dynamically

Usually the size of the cluster doesn't change much over time unless some node fails or more hardware is bought, so the scheduler has to package the jobs across a static number of nodes, this causes that even if the next job only requires one more node than the ones currently available it has to wait for one more to get free, this is not good because either some job has to skip the line in order to maintain those nodes busy for a while, or they get wasted doing nothing until more resources get free.

We propose to make the size of the cluster change in order to allow jobs to run when they require only a few extra nodes, this is possible using VMs and provides some interesting advantages over the use of the physical cluster.

In order to allow the number of computing nodes change in a efficient way, we actually need to make a distinction between the physical nodes and the virtual ones, because they have to be managed in a different way, for instance, it doesn't matter if a VM is being wasted doing no job at all as long as the node hosting it has also other VM that is actually working enough to use most of the resources.

The idea is to fractionize the physical resources by allowing various VMs to share them, this means that we also have to take into consideration the amount of resources free in order to take the decision of creating or destroying a VM. This problem is known to be at least NP-hard [50] even if we don't consider all the resources that can be fractionized.

Implementing an idea like [49] may actually require yet another scheduling system due to all the variables considered, we instead take our first approach to be only fractionize the CPU in order to increase the utilization on per-node basis, that in a global scale should also increase the real utilization of the whole cluster.

We propose the monitoring of the hosts and their resources along with the hosted VMs in order to take decisions concerning the size of the cluster, if we detect overloaded hosts, we may decide to relieve them from some VMs, on the other hand, nodes running below, let's say, 50% of their capacity are eligible for hosting another VM that should start running jobs soon after its creation.

3.2 Formal description

Be $H = \{h_1, h_2, \dots, h_i\}$ the set of computing hosts in the cluster and $V_t = \{v_1, v_2, \dots, v_i, \dots, v_j\}$ the set of virtual machines running at some time t ; since the VMs are the ones executing the jobs and we don't want to waste any resources, we need to have at least one VM running per host, so it must follow that $|V_t| \geq |H|$.

The tuple $h = (id_i, cores_i, ram_i, CpuLoad_i, RamUsage_i)$ describes a host and a the tuple $v = (id_j, cores_j, ram_j, CpuLoad_j, RamUsage_j, name_j)$ describes a VM. The id 's are necessary to uniquely identify either hosts or guests; *cores* and *RAM* are useful information about each (virtual) node that can help take decisions; *cpu load* is a measure of the utilization for either one in the last few minutes, we use it to decide whether or no create a VM and also for load balancing; *ram usage* is reserved for future research, we want to use it together with the cpu load to take decisions; lastly *name* is necessary only in the VM because there must not be more than one host with the same name.

The set H is static because we have no control over it, while the set V_t is dynamic, so $|V_{t1}| \neq |V_{t2}|$ since we can add or remove VMs depending on the utilization of the cluster and the jobs in the queue.

Lastly, the set V_t is updated immediately after the event of creating or destroying a VM; every h_i and v_j are updated automatically by the monitoring software approximately every 15 minutes, this is not necessarily a bad thing, after all, if the job doesn't take 15 minutes to complete, probably it's not worth running on a cluster.

With this, the real utilization of the cluster U_t at the time t can be calculated by adding the independent cpu load measures of each node at some point and dividing them by the total number of available nodes like in equation 3.1.

$$U_t = \frac{\sum_{k=1}^{k=i} CpuUsage_k}{|H|} \quad (3.1)$$

Here the idea is to maximize U_t and keep it high as long as there are jobs in the queue. Lets also notice that the cluster usage reported by the resource manager considers that every node that has been assigned work is running at its full capacity, so that would be the theoretical upper limit of equation 3.1 when every CpuUsage is at its maximum, that happens to be what we are trying to accomplish.

3.2.1 Good cluster usage with FCFS

FCFS as scheduling algorithm is probably the one that gives the worst cluster usage of all the available ones, that's why EAYS backfilling is used to increase U_t . With the use of VMs we aim to keep a high cluster utilization even using FCFS for scheduling. This is achieved with something similar the one of "volunteer computing" described [53], only that the volunteers will be virtual machines launched on underutilized nodes.

Taking into account that for a virtual cluster, the maximum number of nodes that can be requested by a job should be less than equal to the number of physical hosts, and not that of the amount of registered computing nodes (VMs); because there is no point on giving more than that if some VMs assigned to the same job will have to share the same node.

If the results shown in [42] still holds truth, then there is a high possibility that a good number of machines are underutilized while executing a single program at any point in time, then with the maximum number of nodes set as described, even if the next job in line is a wide job (requests many nodes) we are very likely to find some underutilized hosts in which to start additional VMs to let the next job in line start right away.

Unless the cluster is very small, it wouldn't be common to have jobs requesting more than half of the total number of hosts, so the event that no more VMs could be started to run the next job would mean that the cluster usage at that time is very high, most likely with no idle hosts. On the other hand, if two VMs are running in the same host and they have been competing for resources, then one of them would be moved to other host during the load balancing phase, if they are nor really competing, it doesn't matter whether or not they share hosts.

While one of our objectives is to make FCFS a useful scheduling algorithm, it doesn't mean that others won't appreciate the benefits, quite the opposite, any algorithm that can handle events that change the amount of available nodes can be used, so at least the popular ones that limit to regenerate the schedule on such events will get the benefits.

3.3 General solution

The general way we propose for building a virtual cluster on top of the physical one, and which we think is fairly easy, is by using an existing resource manager for the tasks of scheduling jobs and cloud virtualization software. Both must have certain capabilities that we describe here.

Any solution with support for *hardware assisted virtualization* will do, this is necessary to get the most out of both the hardware and the OS used for the cluster; also it is convenient to select something with support for an *hypervisor*, among other reasons, to allow the management of guests without the need of a Graphical User Interface (GUI) and also to gain support for migration.

The **cloud software** to be used must allow to *manage* any number of *hosts* in order to add and remove them from the pool and also be able to disable some in case of any needing maintenance. While an static number of hosts registered would suffice, it's convenient to be able to add and remove them to gain flexibility and even be able to send some outside the local network.

Also the package used has to be able to *monitor* both hosts and guests. The hosts must be watched in order to decide whether or not to allocate any or more VMs in it, most cloud packages with build-in load balancing will do this much, though this is usually only for taking decisions about allocation. Also it's convenient to be able to monitor the guests in order to take decisions concerning load balancing.

There would be very little use of the information retrieved from the resources if we couldn't access it, so the cloud solution must also provide ways of retrieving this information. In the best of the cases, this package would be able to automatically balance the load by migrating the VMs, however we still need to access this data for the reasons described before.

Lastly, the software used should provide support for *VM migration*; while this is not a must, it's very useful in order to fix possible mistakes about VM allocation due to lack of information about node workload or due to the dynamic nature of the jobs running in the (virtual) cluster. Without migration support, the virtual cluster would be limited to (un)allocating guests, loosing flexibility.

Just as cloud software doesn't schedule jobs, the *Resource Manager* (RM) doesn't manage virtual machines, so this means that there is some work that we actually have to do in order to make it communicate with the cloud technology that the RM doesn't know about. It must provide ways for managing the hosts, for instance, it should provide some event-based means to request more resources or liberate existing ones.

Liberate resources : this event would be triggered when there is no work that a particular node/VM can do at the moment or from the last couple of minutes. With this we can shutdown a VM.

Restore resources : the counterpart of the former, an event that should ask to get back the nodes that were liberated when they are needed to run some job. Here we allocate a VM.

This events are needed in order to allows the indirect interaction between the RM and the cloud manager, of course, we need to personalize the way in which such events are treated depending on the information provided or required by both solutions.

In the case that the RM doesn't provide ways for requesting and liberating resources, there is still a way to work with the virtual cluster. A daemon is needed to retrieve information about hosts and guests and allocate more VMs if the utilization is low or shutdown them otherwise; for this to work, the RM must be able to *handle node failure and return to service events*. The optimal way of handling the liberation of a VM would be to ask the RM to stop allocating jobs to it, and then when the usage of the VM falls near to 5% shut it down (the guest is unlikely to fall below that because there are always daemons or services consuming CPU), also we need to tell it to start allocating jobs again if the VM is to be resumed.

The way of actually building this middleware to allows the RM and cloud to interact will depend on the packages used, so in section 3.4 we describe how we actually do it.

3.3.1 Ad-hoc VMs by per node requirements

So far we have seen one way to a virtual cluster, the solution proposed so far will use groups of homogeneous VMs, it wouldn't make sense otherwise since we need a template (probably based on queues) to create the new VMs every time; so for the express queue, probably the templates defines a 2 core VM with only 2 GB of memory (we don't want them to get in the way of the other jobs) while the high memory queue will define VMs with 16 cores and 32GB of RAM.

Other possibility is to create something similar to a external scheduler, something like Maui [46] or Moab. Most RMs will provide ways for checking which is the next job in the execution queue, the idea of a *ad-hoc VM* is to retrieve the requirements of the next job and modify or create the VM templates according to they, for instance we can retrieve at least the number of nodes required, we can enforce a policy of providing a memory estimate for the job too and retrieve that information to make and ad-hoc group of VMs for the exclusive use of a particular job.

In order for this solution to work we need to be able to tell the RM no to assign any job to the nodes, instead we need to create the VMs and *liberate* the job and ask for it to be executed by the indicated nodes, most RM will allow at least the first part, also, requesting particular nodes is usually supported.

While this possibility has the disadvantage that it will include some job start latency (while the VM boots) we actually explored a bit of it and we describe our experience in section 4.3.2.

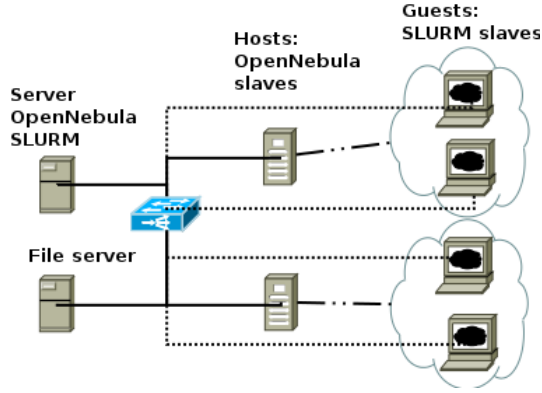


Figure 3.1: In a virtual cluster, the VMs are the ones running the HPC jobs and connect to the network using bridged connections (dotted line).

3.4 Software review

We reviewed some software packages in search for the characteristics described in section 3.3, since there is a number of such packages with their own advantages over one another we review a few popular ones and decided to stay with the Kernel Virtual Machine (KVM) hypervisor, SLURM resource manager and OpenNebula cloud computing. Also for connectivity we use bridged network devices in order to gain access to the local network.

The software that we are using and the basic configuration and roles is shown in figure 3.1. Following we give a small comparison of the different alternatives that we tried and also why we decide to use the aforementioned software and the proposed architecture.

Both the login and file servers must be accessible from the hosts and guests however the later groups doesn't necessarily need to see one another, so in order to access the guests, bridged connection are needed, they can be in the same sub-net or in a different one.

3.4.1 Virtualization technology

There are lots of alternatives for virtualization, the most promising of them: an hypervisor. VMWare provides an hypervisor, however we cannot afford it, so we fall back to free software. There are two good popular alternatives available for the open source community: XEN and KVM.

While XEN is an older solution with lots of interesting features, it was also harder to install, a patched kernel (which is available for many distributions) is needed for it to work and uses lots of concepts like Dom0, DomU, ring, etc. to distinguish between (non) privileged VMs and users. We ran a few tests with XEN, specially while selecting a cloud technology (see 3.4.3) but in the end decided to stay away from it.

Type	Name	Speed	Hardware assisted	Free	Complex terminology	CLI
Emulation	VMWare	Fast	Partial	X	X	X
	VirtualBox	Fast	Partial	✓	X	X
Paravirtualization	XEN	Faster	Complete	✓	✓	✓
	KVM	Faster	Complete	✓	X	✓

Table 3.1: Resume of virtualization technologies.

After XEN, the next we tried was KVM, unlike the former, KVM comes by default as part of the kernel, so we only need to install its tools and it's ready to go, at first the performance wasn't that good, but a little research yielded that it was due to the image format settings, so using a QCOW2 image with metadata preallocation and without write-through cache gives a very good performance [11].

Even though we preferred KVM over XEN or VMWare we still wanted some platform agnostic solution, so we decided to give libvirt a try, it is a very complete package that is compatible with KVM, XEN, Qemu and others, it works through a XML configuration file where we can define the MAC address of the virtual network devices, virtual images to use and drives and other interesting features, however even the simplest configuration file turns out to be fairly big and is not very intuitive.

Working directly from the Command Line Interface (CLI) using KVM or XEN isn't easy because of the amount of parameters that they can receive so a solution that takes care of the dirty job is needed, this is where the cloud technology comes into play, see section 3.4.3.

A resume of the virtualization technologies can be found in table 3.1.

3.4.2 Network

There are a few options for allowing the VMs into the network, for instance let the VMM create something like a *private virtual bridge* that allows the VM into the internet, however the host doesn't see the VM and the VMs can't see each other, so this solution doesn't work for a cluster. Even more, this kind of express configuration is very slow.

It is also possible to create a *private bridge*, this one is a lot faster and allows both host and guests to see each other, however this one doesn't allow to access the internet or guests from other nodes; a workaround for it is configuring NAT in the host, however a better solution exists.

A *public bridge* is the best of the two worlds, it's a permanent connection that allows sharing the physical interface, so it's very fast and allows the hosts and guests to see one another even across nodes, the only issue with this particular bridge is that it doesn't seem to work with wireless cards, however no serious cluster will use a wireless connection, so we went for this one. Both XEN and KVM are compatible with bridged connections.

Type	Speed	Range of the connection		
		Net	Host	Other VMs
Virtual	Slow	✓	X	X
Private	Fast	X	✓	✓
Public	Fast	✓	✓	✓

Table 3.2: From the types of NICs for VMs, the public network is the best.

A resume of the types of network connections for virtual machines can be seen in table 3.2.

3.4.3 Cloud technology

To save ourselves the pain of working at low-level with the VMs and having to implement all the control system needed to administer a handful of images, hosts and guest we decided to turn into the cloud computing technology, this kind of software is excellent for those tasks plus many of them implements contextualization and monitoring facilities of both hosts and VMs.

We started by setting up and testing Nimbus which is a very mature project that integrates with the Globus toolkit, it supports either XEN or KVM, but not both at the same time, setting it up is very difficult mainly due to all the security built around it and it's somehow hard to personalize and administer; besides the very handy *one-click cluster* [31] it also proved other interesting features like run-time requests for limiting the time that the VM will remain running. Overall nimbus turned out to be quite complex and we decided not to use it even with all its interesting features.

The next we tried was OpenNebula, it turned out to be very a flexible and extensible cloud technology, it was very simple to setup (compared to Nimbus) and allow mixing virtualization technologies like KVM, XEN, VMWare, etc. at the same time and the contextualization doesn't require network to work for it uses an ISO image to hold the required files.

OpenNebula also provides a very good set of RPC (Remote Procedure Calls) that makes it a good choice for development, for instance `one.vm.allocate` will try to create a VM, we can decide in which host by calling `one.host.info` to retrieve the status of all the hosts registered in order to make the decision.

There are other interesting cloud systems like Eucalyptus, and also some programmer oriented virtualization frameworks like Usher [39], however there was no time to review them all and OpenNebula turned out to good enough for the proof of concept that we mean to give.

A resume of the cloud computing technologies evaluated is found in table 3.3.

OpenNebula

OpenNebula (ONE) is a somehow popular cloud software out there, many companies, schools and researchers use it because of the amount of feature it pro-

Name	Compatible with				API	Multi-driver
	XEN	KVM	Qemu	VMware		
Nimbus	✓	✓	X	X	X	X
Eucalyptus	✓	✓	X	✓	X	✓
OpenNebula	✓	✓	✓	✓	✓	✓

Table 3.3: Comparison of cloud computing packages.

vides; one of those that use it is the CERN, one of the reasons for them to use it is because of the extensible XML-RPC API that allowed them to automate some process required; a German lab SARA developed their own management console to allow the users to request VMs or upload their own on top of ONE; many others have evaluated many technologies out there and also preferred ONE for similar reasons, like CloudWeavers “*We evaluated basically all the other toolkits on the market, and these were our findings: OpenStack: clean design [...], immature [...], Swift storage is very amazon-like, but slow and cpu intensive. Eucalyptus: the OSS edition is years behind the commercial version, and missing many important things. CloudStack: monolithic, nice looking but terribly complex to debug if something does not work properly. Heavy (Heavy!!!). Difficult to extend. Opennebula was for us much simpler to explore, adapt and customize; core is clean and small, most other things are manageable scripts. We found it stable [...]*”

3.4.4 Resource manager

The Resource Manager (RM) is the one in charge of sending the jobs to the appropriate nodes, it has to monitor the HPC (virtual) nodes and make sure that they are up and working, in case of a node failure, it has to be able to take action, like sending an email to the user or requeuing the job running in the node that failed. With this much it’s obvious that at RM is not a trivial piece of software; since we don’t want to write our own, we have to look for the alternatives.

The first RM we tried was Torque from AdaptiveComputing, it is a industry class open source software with lots of features, the ones that are of interest for us are the ones for changing node states issuing the command `pbsnodes -o nodename` we can mark a node as offline and so it won’t be considered for executing other jobs after finishing the one that is running (if any); also we can return it to service using `pbsnodes -c nodename`. While we can work with Torque with this features, it limits what we can do with it as we can only have the external daemon starting and stopping VMs when it see it appropriate.

We also tested SLURM [30], it very much shares the mentioned node status administration options with Torque, for instance, in order to remove a node we use `scontrol update nodename=name state=DRAIN`, this will stop sending new jobs to the node; in order to make the node available again we issue `scontrol update nodename=name state=RESUME`, this will tell SLURM that

Name	Drain nodes	Extensible	shutdown nodes	Custom APP execution	API
TorquePBS	✓	?	X	X	✓
SLURM	✓	✓	✓	✓	✓

Table 3.4: Comparison of resource managers

it can allocate new jobs to the node again.

Besides those administrative options, SLURM also provides some features for power saving which automatically executes a program in order to decrease its power usage in a custom way, as the node is in power saving mode, no new jobs are allocated to it, so we can take advantage from it to issue the signal to shut down the VM; also nodes in this state are returned to service when there are jobs it can execute by calling another program that we can use to create the VM when it's required.

The interesting power saving features and also some newly introduced support for the EC2 cloud services made SLURM very attractive for the system we were trying to develop, so we stayed with it.

The resume of the evaluation of the resource managers is in table 3.4.

SLURM

SLURM is a cluster resource manager developed by a team from the Lawrence Livermore National Laboratory (LLNL) to manage the workloads from their GNU/Linux clusters, this is also very popular across the free software community, it has even made it into the official repositories of some operating systems like Debian (and its derivatives); it's enough to check the amount of traffic in their user's mailing list and the amount of third party plugins available to have an idea of the amount of people using it for their clusters. Its modularity and helpful community has made it the choice of companies like Intel, The National University of Defence Technology (NUDT), CEA, Swiss National Supercomputer Centre, Barcelona Supercomputer Center and others.

Chapter 4

Results

In this chapter we present the actual implementation of the solution proposed in section 3.3 with the software selection specified in section 3.4; here the most important parts of the software created are shown along with the results of some experiments made using the infrastructure that was made available for this purpose.

While we give more interest in our own software, called Thiao, we also provide some configuration examples for the software used as some times it may affect the performance or it's necessary to take into account some considerations for they to integrate correctly or provide all the functionality required.

4.1 Hardware and software specifications

The specifications of the software and hardware used for both testing and development of Thiao is described here. The example configuration and services running in each node is shown in figure 4.1.

The slave nodes and file servers are from the Sierra cluster at SDSC that is part of the FutureGrid project; this are 8-core nodes with 32GB of RAM memory connected through InfinBand network. The operating system is RedHad Enterprise Linux 6.0 (RHEL6). As some nodes were provided for exclusive use of this project, we ran RHEL6 instead of RHEL5 like the rest of the cluster nodes.

The hosts connect to the NFS server through LDAP which mounts on demand the shared directories, this make a little more efficient the FS as they have 2 NFS servers, it's also necessary to have this NFS directory for the VM images, otherwise it's not possible to configure live migration.

The NFS servers provide user's home directories and also project directories, in the home directory we store our sample codes for testing purposes, in the project directory we store the main VM template image and also the temporal clones for the instantiated hosts along with the necessary descriptor files for management tasks.

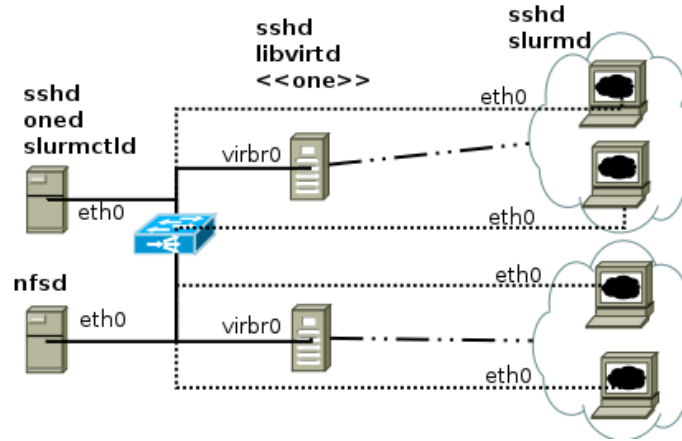


Figure 4.1: Services running in each node, and network interfaces.

The server doesn't need to be very powerful, as it doesn't run VMs nor it stores anything, instead it needs some updated software, as this cloud and virtualization packages depend on new libraries, so RHEL6 was not enough, we used Debian/testing instead as it's fairly stable and provides current versions of the packages with better debugging features than the ones shipped with RHEL6.

While any Operating System can be used for the job execution nodes, and RHEL is one of the preferred ones, we opted for Debian/testing instead, as it's convenient to have the same OS in both the login server and the job execution nodes. Also Debian provides more prepackaged software than RHEL, making it easier to setup.

OpenNebula (ONE) 2.2 was also used, during the development ONE 2.3 was released, but some changes in the way it read the template files broke its backwards compatibility, so we didn't upgrade, furthermore no new features of interest to us were implemented in the new version so the manual upgrade of the package was not worth the effort.

For SLURM we also used version 2.2 since it provides all the features we wanted to test like power saving, node status control and various scheduling algorithms, at the point of testing; in the latest development version, support for Amazon's EC2 services was being added, this was one feature that we wanted to test, however the support was still in the planning phase; having some direct communication with Moe Jette (SLURM's architect) we discovered that the main difference with our method was that EC2 allowed to change the IP of the newly created hosts which isn't really necessary for our solution.

With all this properly known and configured we got an appropriate environment to develop the solution proposed in 3.3 and also run some test to compare the performance upon finishing.

4.2 Architecture of Thiao

As first result we developed a software that we called Thiao, this word doesn't have an special meaning, it was extracted from one of the text from Aleister Crowley; this software was necessary because no other solution for virtual clusters existed at the time of the research/writing.

Thiao is the software created to serve as a middleware for both SLURM and OpenNebula in order to create a virtual cluster, as of the point of writing it's a solution dependent of both pieces of software, our objective is to demonstrate that it's possible to create a virtual cluster for HPC and evaluate it's advantages over a normal one.

Thiao is divided in various pieces, some standalone and some complementary, here we describe each one of them and show the way they were implemented and how they work towards our objective.

Upon finishing all the different blocks of software described in the following sections we were finally ready to run some experiments.

4.2.1 Basic VM management

While we can manage the VM on our own without the intervention of the RM we still prefer to let it participate in the process by sending requests for resource allocation or liberation, this is convenient because this way we don't need to be pooling the job queue and we still can liberate the VMs when necessary based on the status of the cluster.

SLURM can make requests for allocation and liberation of resources through some power saving features that has built in, this automatically calls the command defined as `ResumeProgram` in `slurm.conf`; `ResumeProgram` is called with the host names of the VMs that it is requesting as parameter, this program can do whatever we want, wakeup the indicated host from suspend state, increase their CPU frequency, etc.

In order to enable the power saving mode we need to indicate the path for the resume and suspend programs in the `slurm.conf` configuration file as follows (this will use Thiao's resume and suspend programs):

```
#/etc/slurm-llnl/slurm.conf
SuspendProgram=/opt/thiao/bin/Suspend
ResumeProgram=/opt/thiao/bin/Resume
ResumeTimeout=180
SuspendTime=60
```

When there are jobs in the queue and some nodes in power saving mode, SLURM will call the program called `Resume` that comes with Thiao with the list of VM names to be launched as parameter, this is because it was set as the `ResumeProgram` in the configuration file. `Resume` then looks into the `/opt/thiao/examples` directory for a VM template file with the same name of the host to be launched, then sends an XML-PRC request to OpenNebula to

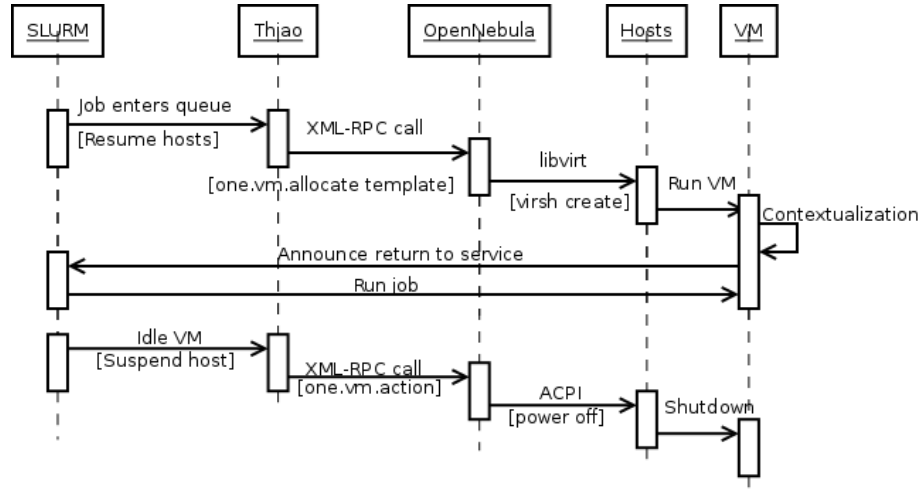


Figure 4.2: Sequences for starting and shutting down VMs through SLURM's power saving features.

allocate the indicated VMs and have it start executing jobs; we can see this sequence in the first part of figure 4.2.

The VM template file indicates the image to use, whether to clone it or not and the contextualization variables to be used like IP address, number of network interfaces, number of virtual CPUs, size of the RAM memory, DNS servers and optionally files to send along to the VM among with some more information, an example of one looks like this:

```

NAME    = "fg0"
vCPU    = 3
CPU      = 1
MEMORY  = 3000
DISK = [
    source = "/N/project/opennebula/img/d-testing.img",
    target = "hda",
    driver = "qcow2",
    readonly = "no",
    clone = "yes" ]
NIC = [ NETWORK = "fg1" ]
CONTEXT = [
    gw      = "$NETWORK[GATEWAY, NAME=\"fg1\"]",
    netmask = "255.255.254.0",
    ip      = "192.168.11.",
    dns     = "$NETWORK[DNS, NAME=\"fg1\"]",
    files   = "/etc/slurm-llnl/slurm.conf /opt/thiao/examples/mio.sh
              /opt/thiao/examples/mictx.sh /opt/thiao/examples/hosts",
    target  = "hdb"
]

```

If a VM is idle (no jobs assigned to it) for more than `SuspendTime` seconds, SLURM calls another Thiao module called `Suspend`, that as the other, receives a list of unused VMs and asks OpenNebula to shutdown them, so this complements `Resume` as we can see in the second part of the sequence diagram shown in figure 4.2.

Since the resource manager doesn't know about the physical hosts and doesn't know that the execution nodes are virtual, it will keep on asking to resume VMs as long as there are job in the queue that can be run by allocating one host, however Thiao is aware of the hosts and the fact that they can hold a limited number of VM, therefore it can decide no to give more VMs if the utilization reported by OpenNebula is over the required limit, we propose to start a VM only if the utilization of the node is below 60% of it's capacity, otherwise it's very likely that the new VM would compete for resources with the other and slow down the execution of both.

The way the Remote Procedure Call (RPC) works is by requesting the execution of a registered service with a set of parameters formatted in eXtensible Markup Language (XML), thereof XML-RPC call. OpenNebula 2.2 provides over 30 XML-RPC calls for various tasks, all the calls receive as first parameter a session string to authenticate the user issuing the call.

For allocating and liberating VMs we are interested in the calls with the signatures `one.vm.allocate` and `one.vm.action`, the second parameter of the former one is the VM template string indicated before, OpenNebula 3.0 also allows a third parameter to indicate which node will host the VM, we plan to update the code for that in the near future. The second call performs actions like

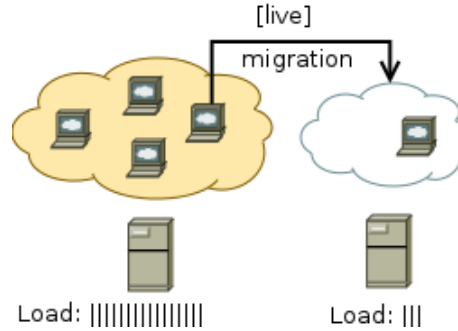


Figure 4.3: Load balancing achieved by migrating VMs from one host to another.

shutdown, restart, suspend, etc. the action to execute is the second parameter and the third parameter is the ID of the VM in which to perform the action. Both calls will return an XML formatted string with the result of the operation and a string indicating the errors if any.

With this now we are able to automate the creation of virtual nodes to execute jobs in the cluster.

4.2.2 Load balancing

As we just explained, currently we don't have control over where each VM will be hosted, however, OpenNebula with KVM and LibVirt provide facilities for live VM migration. Live migration is known to be a fast and efficient way of doing load balancing [10] as the downtime of the VM is minimal (between 210ms and 60ms depending on the size of the VM and the work being done).

A module called Balancer was implemented, as it's name suggests it's purpose is to balance the load across the nodes of the cluster, it is intended to be executed automatically using some service like *cron* probably every hour. Balancer is implemented completely independent of SLURM, however it depends on OpenNebula as it retrieves information about the hosts and the guests from it.

Lets take as example the figure 4.3, for whatever reason we ended up with many VMs executing in one host, and now it's running close to 100% of it's capacity, there is also other node that is underutilized; as depicted in figure 4.4, balancer will retrieve the information about all the registered hosts and VMs, in an event like the one described, Balancer will ask for a live migration of a VM from one host to another, increasing the utilization of one and leaving some extra resources at the disposal of the VMs of the other host (which may actually use them immediately! in case that they were competing for resources).

As we see in figure 4.4 Balancer also issues XML-RPC calls to the OpenNebula server, this are `one.hostpool.info`, `one.vmpool.info` and `one.vm.migrate`, the two `*pool.info` procedures return a list of hosts or guests with their status which includes number of processors, size of the memory, the average utilization

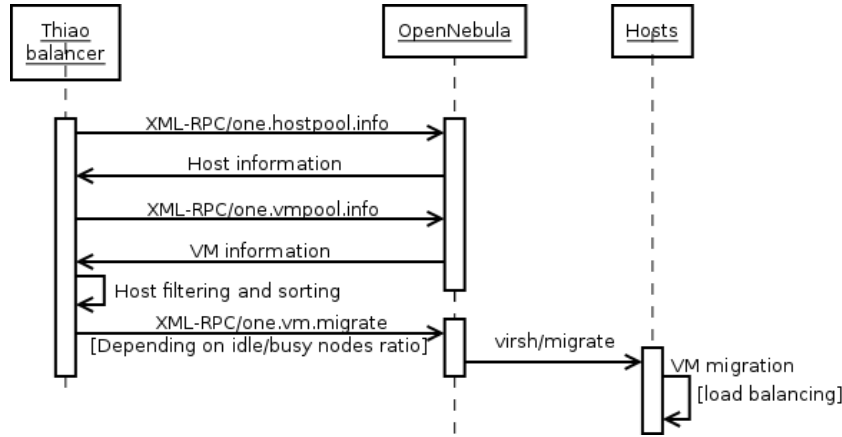


Figure 4.4: Host and VM information gathering in order to balance the load.

and some more information.

Having the status of all the hosts and guests we store them in lists, drop the ones that are failing or out of services and then sort them based on their load (utilization), having this we can expect the ones at the head of the list to be the most busy ones and the ones at the tail the least used, if the next node from the head of the list is running more than one VM and the next from the tail is running below the desired utilization level, Balancer issues `one.vm.migrate` to OpenNebula including as parameter the destination hosts (tail of the list), the ID of the VM to migrate and also tells it to perform a live migration, the output of this request is the result of the operation; after this we drop the head and tail of the list and repeat.

Executing Balancer automatically, we can keep the load of the nodes balanced, however at this point we don't consider yet the memory usage of the nodes during the balancing process, this could be a problem in the event that the size of the VMs surpasses the amount of memory of the hosts as it would be forced to use paging in order to maintain its VMs running which could lead to some slowdown in the execution.

The live migration of VMs is a feature available in many hypervisors, it allows to move a VM from one host to another without stopping the computation, so the penalty for the migration is very small, specially when there is a high speed connection between the hosts; that's another one of the interesting features of virtualization.

In order to enable live migration it is necessary that both, the source and destination hosts, see exactly the same OS image in the same path, this is easily achieved by configuring some shared directory, for example with the Network File System (NFS), a shared directory is part of the standard configuration of most if not all clusters.

It is possible to migrate VM from one host to another even without a shared

directory, but the process is very slow as it saves the VM to the disc, then copies the resulting image to the destination and then tries to execute it again, this process is proportionally long to the memory size of the VM and it is very likely to make the RM think that the node failed as it wouldn't respond for a few minutes while the migration is in process; the RM then will cancel the job and queue it again using other nodes.

With this load balancing we can increase the overall utilization without having to necessarily create more VMs on the nodes, instead we move the existing ones around to achieve the same thing.

4.2.3 Flexible scheduling

By default, the EASY scheduler always tries to execute the jobs in the same order as they arrived (FCFS) and then uses backfilling to fill the gaps in the schedule; mixing both the power saving features that asks for VMs and the EASY backfilling we automatically get a more flexible scheduler that will try first to launch more VMs in order to run the next job in the queue, and only if no more (or enough) VMs were given will backfill other jobs (probably bigger than the ones that could have run otherwise).

This is exactly what happens in our current test environment using OpenNebula and SLURM, the ultimate goal is to execute the jobs in FCFS order as much as possible without wasting the cluster (which is usually the result of using FCFS as scheduling algorithm) by extending and decreasing the size of the cluster depending on the demand of resources and the utilization of the nodes.

Other possibility is to ease the use of *express queues* registering some dedicated VMs on the least busy nodes, and also the sharing of consumable resources like licenses; also it would be easier to schedule dedicated hardware like GPUs without leaving the hosts that have those resources idle until some job requests them which is one of the things that happens, keep the node idle until a job requesting GPUs enters the queue and execute it immediately, or run jobs in the node and when a job that needs GPU enters the queue have it wait until the node gets free.

4.2.4 Advanced management

We also implemented a light version of the ad-hoc VM mode described in section 3.3.1, this was intended to test how convenient would the advanced management of resources and virtual nodes was, it turned out to be quite difficult and much more dependent on the resource manager. Only compare the sequence diagram 4.5 with the one using the power saving features 4.2.

For the per job VM mode to work correctly, we need a far more elaborated middleware, it actually needs to closely resemble an external scheduler that tell the RM what to do and when like Catalina [55], MOAB [28] or Maui [46], otherwise the VMs are not used as expected.

The way we developed this advanced management program was by creating our own submit command called `tsub` that is used instead of `sbatch`, `tsub` will

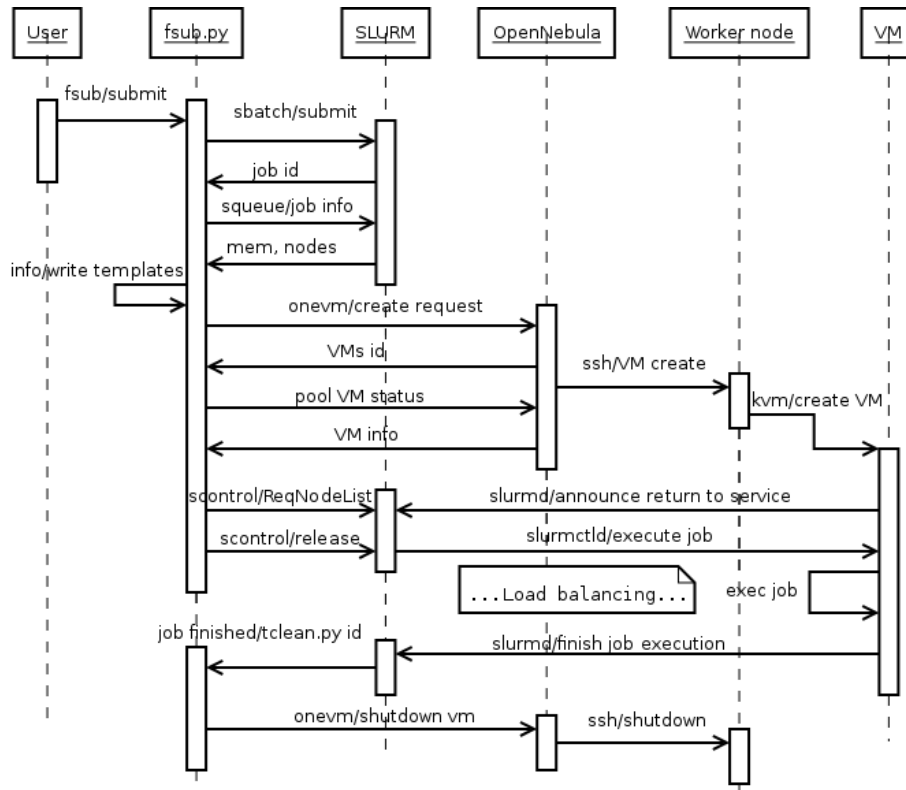


Figure 4.5: Services running in each node, and network interfaces.

receive the job file and internally submit it in the name of the user, the resume of steps performed is as follows:

1. The job is submitted internally using `sbatch` and save the job id for future use.
2. We retrieve the job requirements pooling the job queue through `squeue -o '%5D %7m' -j <job id>`.
3. As many VM templates are created as hosts requested by the user with the appropriate memory limits of the job.
4. Launch the VM defined previously.
 - (a) Store the VM ids and link them to the job id for which they were created in our local database.
5. Wait a little for the VMs to boot (or pool the VM database to know their status).
6. Set the nodes in which the job can run using `scontrol --ReqNodeList <VM hostname list>`.
7. Let the job start running using `scontrol release <job id>`.
8. Prevent any job from starting running in the ad-hoc hosts by changing their status `scontrol nodename=<VM> state=draining`.
9. Once the job finishes, a program that cleans the environment, called `tclean.py`, is automatically executed.
 - (a) Shutdown the VMs.
 - (b) Clean the VM and job ids from the local database.

If we don't limit the nodes in which the job can run, we may end up shutting down an incorrect VM, messing our environment, also we need to "drain" the nodes in order to prevent any job from executing in them after the job for which they were created finishes. As we can see, this is quite complex and we don't even do scheduling yet!.

4.3 Experiments

In order to test Thiao in a real environment, we ran a few parallel jobs in the local cluster, here we show the logs from SLURM and OpenNebula and evaluate what was going on inside each of this middlewares than now collaborate to run efficiently this jobs as to see what we can expect from this virtual cluster.

```
ismael@kriemhild:~$ sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
virtual-*  up      3:00:00    17  down* fg[4-6,8-10],vm[2-12]
virtual-*  up      3:00:00     5  idle~ fg[0-3,7]
ismael@kriemhild:~$ onevm list
ID      USER      NAME  STAT  CPU    MEM      HOSTNAME      TIME
ismael@kriemhild:~$ squeue
JOBID PARTITION  NAME      USER  ST      TIME  NODES NODELIST(REASON)
ismael@kriemhild:~$ █
No VMs running
No jobs in the queue

root@kriemhild:/var/log/slurm-llnl# tail slurmctld.log
[2011-11-10T11:44:01] read slurm_conf: backup_controller not specified.
[2011-11-10T11:44:01] Running as primary controller
[2011-11-10T11:44:02] Power save mode: 5 nodes
root@kriemhild:/var/log/slurm-llnl# █
```

Figure 4.6: There are no jobs running or in the queue, all the VMs in power saving mode are in state `idle~`.

4.3.1 Powers saving mode

We start with no jobs in the queue as we can see in the status output shown in figure 4.6, since there is nothing requiring resources, all the nodes are put in power saving mode (bottom) identified as `idle~` (top) as opposed to `idle` (not shown) when are up but doing nothing, here there is no point on letting the VM run if it will be idle anyway; here we can also see some `down*` nodes (top), this status means that the nodes failed or were shutdown externally, so SLURM won't try to allocate jobs in them until they are started manually (only the first time needed).

In the figure 4.7 we can see what happens when a job is submitted; for starters, it's not necessary to modify the job's description file as we can see in the middle of the figure, so it's a seamless migration from the user's point of view; if we start with no jobs in the queue like in figure 4.6, upon submitting a job with `sbatch` (top) SLURM will wakeup (request) some VMs through Thiao's `Resume` program (bottom), so OpenNebula will allocate some (top).

Next we tested how it worked in the presence of big and wide and narrow jobs, starting with the five nodes in power saving mode shown in figures 4.6 and 4.7 we submitted a few 4 node jobs to be executed for which VMs `fg0` to `fg3` were launched to run them as seen in figure 4.8, next we submitted a few one node jobs using `sbatch`, after a few seconds SLURM decided to try to launch the remaining VM to backfill the job and start executing it right away sharing the node with the other VMs. In this particular case, the new VM doesn't compete for resources with the other VMs that were already running since it only used one core.

Overall the results were as expected, now we have a fully virtualized cluster for HPC workloads, however it is still possible that some VMs compete for resources, in those cases the execution time of the jobs increases a little, however, even if the parallel process in one computing node finishes before its siblings the

```
ismael@kriemhild:~$ sbatch /pub/ismael/o5m.sh Submit a job
Submitted batch job 284
ismael@kriemhild:~$ squeue
JOBID PARTITION NAME USER ST TIME NODES NODELIST(REASON)
284 virtual-f o5m ismael CF 0:03 4 fg[0-3] Assigned some (dormant) nodes
ismael@kriemhild:~$ onevm list
ID USER NAME STAT CPU MEM HOSTNAME TIME
221 oneadmin fg-0 pend 0 OK 00 00:00:08
222 oneadmin fg-1 pend 0 OK 00 00:00:08
223 oneadmin fg-2 pend 0 OK 00 00:00:07
224 oneadmin fg-3 pend 0 OK 00 00:00:06
ismael@kriemhild:~$
#SBATCH --time=30
#SBATCH --nodes=4 Regular MPI job
#SBATCH --ntasks-per-core=1
#SBATCH --job-name=o5m
cd /pub/ismael/
time mpirun -mca orte forward job control 1 ./o5m 180
/pub/farfan/o5m.sh [+][R0] 1,1
[2011-11-10T13:33:11] Power save mode: 5 nodes
[2011-11-10T13:44:11] Power save mode: 5 nodes
[2011-11-10T13:46:51] _slurm_rpc_submit_batch_job JobId=284 usec=13266
[2011-11-10T13:46:51] sched: Allocate JobId=284 NodeList=fg[0-3] #CPUs=4 Boot this 4
root@kriemhild:/var/log/slurm-llnl#
```

Figure 4.7: Job files doesn't need modifications, upon submitting a normal parallel job, some VMs are launched to run it.

```
ismael@kriemhild:~$ sbatch /pub/ismael/o5m.sh Submit some 1 node jobs
Submitted batch job 315
ismael@kriemhild:~$ onevm list
ID USER NAME STAT CPU MEM HOSTNAME TIME
230 oneadmin fg-0 runn 183 2.9G s80r.idp.sdsc.f 00 00:14:38
231 oneadmin fg-1 runn 210 2.9G s81r.idp.sdsc.f 00 00:14:38
232 oneadmin fg-2 runn 209 2.9G s80r.idp.sdsc.f 00 00:14:37
233 oneadmin fg-3 runn 208 2.9G s81r.idp.sdsc.f 00 00:14:36
235 oneadmin fg-7 pend 0 OK 00 00:00:18
ismael@kriemhild:~$ sinfo
PARTITION AVAIL TIMELIMIT NODES STATE NODELIST
virtual-* up 3:00:00 1 alloc# fg7 We have to wait a little while it boots
virtual-* up 3:00:00 17 down* fg[4-6,8-10],vm[2-12]
virtual-* up 3:00:00 4 alloc fg[0-3]
ismael@kriemhild:~$ squeue | tail
307 virtual-f o5m ismael PD 0:00 4 (Priority)
308 virtual-f o5m ismael PD 0:00 4 (Priority)
309 virtual-f o5m ismael PD 0:00 4 (Priority)
310 virtual-f o5m ismael PD 0:00 4 (Priority)
311 virtual-f o5m ismael PD 0:00 4 (Priority)
312 virtual-f o5m ismael PD 0:00 4 (Priority)
314 virtual-f o5 ismael PD 0:00 1 (Priority)
315 virtual-f o5 ismael PD 0:00 1 (Priority)
301 virtual-f o5m ismael R 1:21 4 fg[0-3]
313 virtual-f o5 ismael R 0:36 1 fg7 Backfilled !
ismael@kriemhild:~$
```

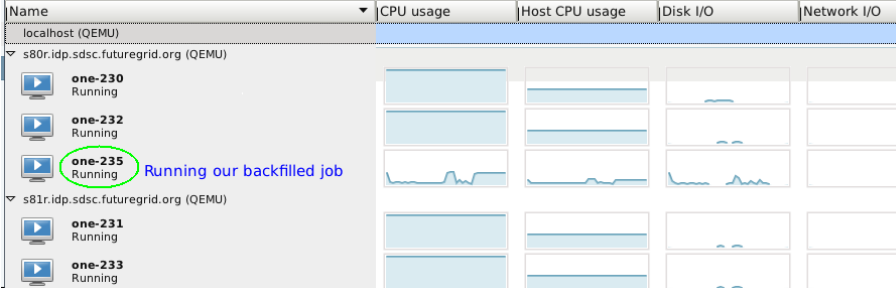


Figure 4.8: The jobs can be backfilled on the running VMs or starting a new one.

```

223 oneadmin fg-2 pend 0 0K 00 00:00:07
224 oneadmin fg-3 pend 0 0K 00 00:00:06
ismael@kriemhild:~$ squeue
JOBID PARTITION NAME USER ST TIME NODES NODELIST(REASON)
284 virtual-f o5m ismael R 2:10 4 fg[0-3]
ismael@kriemhild:~$ onevm list
ID USER NAME STAT CPU MEM HOSTNAME TIME
221 oneadmin fg-0 runn 0 0K s80r.idp.sdsc.f 00 00:02:18
222 oneadmin fg-1 runn 0 0K s81r.idp.sdsc.f 00 00:02:18
223 oneadmin fg-2 runn 0 0K s80r.idp.sdsc.f 00 00:02:17
224 oneadmin fg-3 runn 0 0K s81r.idp.sdsc.f 00 00:02:16
ismael@kriemhild:~$

[2011-11-10T13:11:11] Power save mode: 5 nodes
[2011-11-10T13:22:11] Power save mode: 5 nodes
[2011-11-10T13:33:11] Power save mode: 5 nodes
[2011-11-10T13:44:11] Power save mode: 5 nodes
[2011-11-10T13:46:51] _slurm_rpc_submit_batch_job JobId=284 usec=13266
[2011-11-10T13:46:51] sched: Allocate JobId=284 NodeList=fg[0-3] #CPUs=4 Nodes started when needed
[2011-11-10T13:49:01] error: Nodes fg[0-3] not responding Wait for them to boot and contextualize
[2011-11-10T13:49:01] Job 284 launch delayed by 130 secs, updating end_time Some times takes too long to start
[2011-11-10T13:49:02] sched: _slurm_rpc_job_step_create: StepId=284.0 fg[1-3] usec=437 Gave up on VM 0
root@kriemhild:/var/log/slurm-llnl#
ismael: bash

```

Figure 4.9: Sometimes the VMs takes too long to “return to service” and others are used instead.

node isn’t likely to be wasted as it might be hosting more than one VM at a time making use of the dead cycles.

4.3.2 VM on per job basis

Lastly, we also did some experiments launching VM on per job basis, that is, modifying the VM template files depending on the requirements of the jobs as this looked quite promising; we ran, however, with some issues: users are very likely to overestimate the runtime of their jobs, which is the one mandatory parameter for most resource managers, they would be even less likely to provide good memory requirements for their jobs which is not a mandatory parameter.

As is expected, the boot time for the VMs is not always the same, many things in the way can make it faster or slower, for example the load of the host booting the VM, the network or I/O load of the file server as in needs to clone (copy) the VM image before booting. This variability of the boot time can lead to some problems: when we start some VMs to run a particular job, this VMs are assigned to it and given some time to boot, if the amount of time it takes the VMs to boot is longer than usual it could be marked as **down** or **not responding** and therefore not used.

In figure 4.9 we can actually see that for a job that required a varying number of nodes (it required either 3 or 4) first we tried to launch the upper limit of nodes requested, but one of them didn’t boot in time so SLURM gave up waiting for it and only 3 were used to run the job.

While the wait time before considering a VM as not responding can be configured, this extra time still counts as part of the runtime of the job for the which the VM was created as we can see in figure 4.10, though this is part of the results from the power saving mode, this is still what we can expect from the “per job VM” policy. In the power saving mode this extra time counts only

```

301 virtual-f o5m ismael CD 1:28 4 fg[0-3]
302 virtual-f o5m ismael CD 1:24 4 fg[0-3]
303 virtual-f o5m ismael CD 1:24 4 fg[0-3]
304 virtual-f o5m ismael CD 1:24 4 fg[0-3]
305 virtual-f o5m ismael CD 1:23 4 fg[0-3]
313 virtual-f o5 ismael CD 2:51 1 fg7
314 virtual-f o5 ismael CD 1:21 1 fg7
315 virtual-f o5 ismael CD 1:21 1 fg7
307 virtual-f o5m ismael PD 0:00 4 (Resources)
308 virtual-f o5m ismael PD 0:00 4 (Priority)
309 virtual-f o5m ismael PD 0:00 4 (Priority)
310 virtual-f o5m ismael PD 0:00 4 (Priority)
311 virtual-f o5m ismael PD 0:00 4 (Priority)
312 virtual-f o5m ismael PD 0:00 4 (Priority)
306 virtual-f o5m ismael R 0:37 4 fg[0-3]
ismael@kriemhild:~$

```

The time required to start the VM is discounted from the job's requested time...

Figure 4.10: The time that one VM takes to start is counted as part of the job's real runtime.

for the first job running on the VM, the rest doesn't get penalized because the VM is already running.

This variation in the boot time might be negligible for big jobs, it could also be a problem if we have to wait 10 minutes while the VMs boot just so that the job fails to start due to an user error in the job description file.

Also, another problem we ran into with this solution is that the resources are not always used as expected, to begin with, we have to specify the base features of the computing nodes in the `slurm.conf` file, this is what the nodes are registered with when they are sleeping. If we define less resources than the ones defined in the configuration file, SLURM sometimes doesn't use the nodes because they mismatch, so we need to define the minimum number of resources that will be available for the VMs. On the other hand, the VMs not always have time to report the real number of resources that they really have, so the job ends up running in a VM with lots of resources but only using a few of them, we specially noticed this while setting the number of cores to be used.

This solution, however, is quite promising since not only allows us to share the resources as well as the nodes, but also is another way of job "processor affinity" [57], as this is another problem in the table, specially because some users might request only one core in order to move quickly trough the queue and then launch lots of threads effectively using more resources than the ones requested. If we run this kind of jobs in one core VMs, there is no way that the user gains anything from doing such things.

4.3.3 Performance

In order to evaluate the performance of the solution proposed, a set of runs were prepared, we start with a fixed amount of jobs that are queued all at once in random order, we start by measuring the amount of time it takes the cluster to run them all in bare metal, then we continue by allowing each node to run one, then two, then n VMs at once and each virtual machine to run a single job at a time from the pool of jobs.

The workload consists on 107 configurations of the AIRES cosmic rays sim-

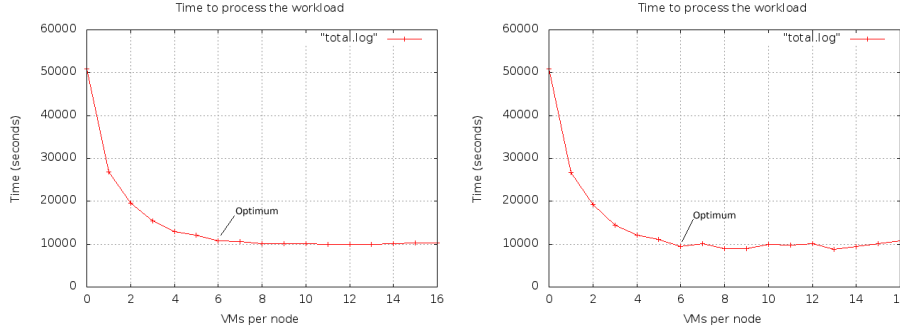


Figure 4.11: Total time to process the 107 jobs workload: a) Incremental VM creation b) All VMs available from the beginning.

ulator, in dedicated bare metal each job runs for between 20 and 40 minutes, this configurations consists very much on the elements of the periodic table of elements to prevent the same from running more than once.

A total of 33 experiments were performed, for each of this 33 one batch of 107 jobs was sent in random order, giving a total of 3531 jobs successfully run from beginning to end, no more experiments were executed due to time reasons, because in contrast with simulations, here we actually have to launch the experiment by hand, recollect the data and run it again in case of failure, each one of this successful experiments correspond to one point in the graphics shown in this section.

In most configurations the user will be able to request full nodes even if only one core is needed, this is allowed because there is no way to know in advance how many cores will be used, so here we take the same approach.

The first experiment consists on queuing the batch of jobs and then gradually creating virtual machines every 4 minutes until the specified limit of VMs for that experiments is reached, each node will boot 1 VM at a time, since we have 4 nodes 4 new VMs will be available to run jobs every 4 minutes.

For the first experiment, as the amount of VMs per node increases least performance gain is noticed (figure 4.11a), that comes from the fact that we are waiting 4 minutes between each creating of VMs so by the time the 8th batch of VMs is created 32 minutes will have passed, time during the which a few jobs will have finished executing and many others will be very close to; for the experiments past 12 VMs per node the last VMs to be created hardly get to run one job because the queue is emptied before they finish their first job.

The average job execution time for the first experiment (figure 4.12a) is a little misleading, for the experiments of less than 8 VMs per node, there is hardly any competition for resources at all, however after 8 VMS per node the jobs start fighting among them for the resources (mainly processor cores), not only that, but also the time it takes each job to finish gradually increases with the amount of VMs created and thereof the the amount of jobs executing at the

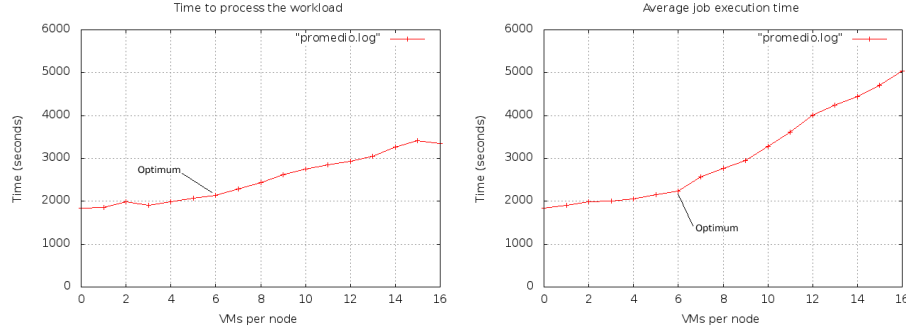


Figure 4.12: Average job runtime: a) Incremental VM creation b) All VMs available from the beginning.

same time in a single machine.

The second run was starting the the given amount of VMs before queuing the jobs, this way the nodes will start working at full capacity in just a few seconds after starting sending the jobs instead of progressively using the resources until the upper limit is reached.

As we can see in figure 4.11b the total time to finish processing all the workload is lower than in the first experiment in many cases, that's not only because of the immediate availability of the VMs but also because less amount of jobs fail to begin execution, compared to the first experiment where up to 15 jobs needed to be automatically requeued because they couldn't start running at all, so less time is wasted in at least 2 aspects.

In figure 4.12b we can see the average job execution time, from 0 to 6 VMs per node we notice that both graphs are pretty much the same, however from that point on the average job execution time for the second experiment increases much more, that is because in the first experiment the competition for resources is gradual while in the second it presents itself from the very beginning affecting all the jobs and not only the middle to last ones.

From this results we can conclude what is quite obvious: we gain from making use of available resources but the most we cross the line of overloading the nodes the least is the gain because it affects noticeable the jobs that are being run in a cluster in order to gain performance in the first place. Also for this particular kind of jobs the optimal number of VMs per node is 6, which gives a total of 7 processes running at the same time and leaving the 8th processor available to take care of the random process that each VM and the host needs to run (like login systems, session managers, etc.)

Lastly but not least important, from the experiment where a total of 1 VM per node was available from the beginning we calculated the average time the jobs spent running in the physical nodes and the VMs; a total of 53 jobs ran in VMs and other 55 ran in the physical nodes, the average runtime of the jobs that ran in VMs was 1995 seconds while the average of the jobs running on

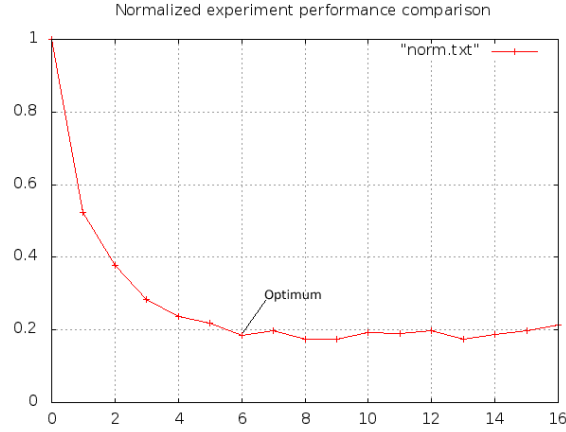


Figure 4.13: Normalized experiment comparison points, percentage of time spent to finish the experiment in base of VMs per node.

physical nodes was 1986 seconds, the ratio virtual/real was 1.0046, so the VMs are about 0.46% slower than the physical nodes for this particular kind of jobs, lets remember that this ratio increases with the number of VMs running (which is bad).

In figure 4.13 we can see the normalized performance of the experiments where all the virtual nodes are available from the beginning, so for example the experiment where only one VM was created per node finished in 52.45% of the time spend in the base experiment, for 2 VMs per node it spent 37.78% of the base time and so on with a minimum close to 17% at 8, 9 and 13 VMs per node.

Measuring the energy utilization of the nodes is beyond the scope of the work, however if we could expect some energy usage similar to the one in figure 4.14, we can see that even if the node uses about 2.5 times more energy while working at full capacity it is actually a good gain considering that it would finish about 5 times faster (20% of the base time, see figure 4.13), so we not only gain in time but also in energy efficiency.

We have proved that the automatic creating of virtual nodes to run jobs in a cluster actually pays off for the kind of jobs that doesn't make an efficient use of the resources allocated, not from the point of view of a single job, but from the point of view of the whole number of jobs in the waiting queue.

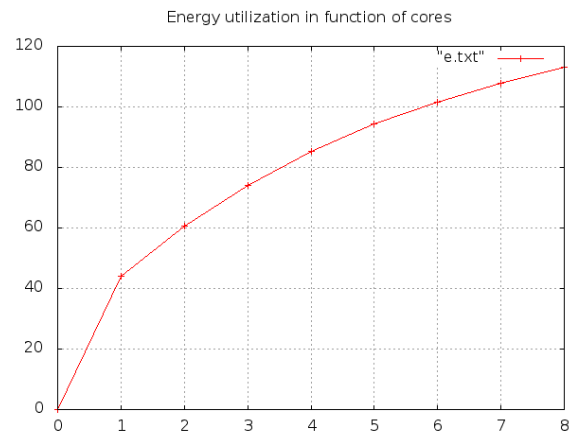


Figure 4.14: Possible energy usage in function of the number of cores working.

Chapter 5

Conclusions and future work

We believe that the software developed (Thiao) is a great tool that serves as a proof of concept that the use of virtualization for HPC workloads is actually possible without too many hassles, and we hope it will motivate others to actually start using it seriously, at least in the middle-range clusters.

However Thiao is not the panacea that will cure the problems of all the clusters, like the seemingly mutual exclusion between Quality of Service, fairness and utilization; though we hope it will make it easier to try to solve this problems without having exclude some of them.

Even though the objective of Thiao is to automate the management of the virtual machines there is still an amount of work that requires supervision from a cluster/system administrator and/or the support staff, for instance the configuration cannot be automated nor are we aware of a tool that does such a thing, there must be always someone with a good knowledge of the cluster's infrastructure and the needs of the group to configure it specially since the quantity of possible configurations varies greatly, also problems related to services, like home directory not being mounted, etc. must be taken care of in a cluster-specific way.

Here we present the resume of our achievements and also the set of problems that we still need to work on in order to have a software fit to be used in production systems.

5.1 Virtualization and resource managers

The performance of the Virtual Machines is such that nowadays they are fit to us in High Performance Computing, however very few people has seen any advantage on doing that, mainly because the creation of the VMs should be manual because no framework for integrating the VM management with the job scheduling existed.

HPC computing since its beginning has been performed over a physical hardware with a mostly static number of nodes (except when there is a hardware failure or new equipment is acquired), so the Resource Managers are not really developed with the thinking that they must also manage the hardware resources and not only assign jobs to them.

Since there exist this gap between these two problems, HPC and virtualization, a few people have been trying to close it, proposing the use of VMs for HPC workloads, however, no framework seemed to exist yet to facilitate this use, many of the researchers have to develop their own management software and scheduling algorithms and don't actually integrate with existing schedulers or cloud management systems.

For instance, many use frameworks like Usher that allows them low level access to virtual machine creation, however, this package is not even a standard software to use, it has not been updated in years and leave lots of work to the programmer in order to give a fine grained VM management framework that is not really necessary.

The software we developed is meant to close the gap between workload resource managers and the more recent cloud systems, allowing the cooperation of two popular packages, the SLURM resource manager and the OpenNebula cloud system which originally were developed for completely independent purposes, but now they can cooperate.

Thiao allows the unattended management of a virtual cluster that grows and shrinks in size when it's necessary, for instance, if a set of nodes are not working to their full potential, more VMs are created and sent to those lazy nodes in order to start running wide jobs sooner and increase the utilization of the nodes.

The dynamic increase and decrease of execution nodes gives a great scheduling flexibility, for instance, with this it is possible to schedule faster the jobs that require specialized hardware like GPUs, while before it was necessary to wait for the node to get freed or to modify the whole schedule, now the job requesting such resources can start running almost immediately just by starting a new VM with access to those particular resources.

In the event that one machine has a shareable resource, but doesn't seem to have enough resources to host one more VM, it is still possible to fix the problem simply by migrating one VM from the node to other node, this is not possible when we are working directly with the bare hardware.

Also Thiao has a module that allows to do load balancing across the nodes by migrating VMs between them, this is necessary because the extra VMs are meant to increase the utilization of the nodes, but if we just keep creating VMs the nodes are overloaded and the jobs will interfere with each other; we don't want the jobs to take longer to finish just because we packaged lots of jobs in them, what we want is to keep the usage high without slowing down the execution time of the jobs.

Thiao is a free and open source software and can be found at github.com/scarmiglione/thiao.

5.2 Future work

There are still a few issues that we are aware of that need to be solved to make Thiao a better software, since so far it's more like a prove of concept to shown that it is possible to force the collaboration of RM and cloud systems, we left this problems for further research.

For instance we are only considering one resource when deciding whether or not to allocate more virtual machines, it is the CPU, the CPU is the bottleneck of modern computers, and we want to keep it busy for as long as possible without affecting too much the performance of the running jobs.

Besides the CPU we need to take into consideration also the amount of memory RAM available in the machines, if we try to host two or more virtual machines in one node that together sum more memory than the one available in the host, it will be forced to use paging in order to keep all the VMs running, this could negatively affect the processing speed of the jobs, specially if they access random parts of the system's memory.

Other issue that we haven't been able to solve is the job suspend and resume, mainly using the Berkeley Lab Checkpoint/Restart (BLCR) kernel module, we need a way to pause jobs in order to allows jobs with greater priority to run immediately when not enough resources to start the VMs are available, however the BLCR kernel module is not compatible with the Linux version we are using, so we are right now between two choices, use an older OS and setup BLCR, or use some recent OS to do actual research and don't job suspend/resume yet; we got for the later for now since this "kick low priority jobs to run high priority one immediately" policy is used only in very particular cases, most of the time will just stop taking jobs from the lower priority queues until the high priority one gets empty.

Bibliography

- [1] M. Astley, D. Sturman, et al. Customizable middleware for modular distributed software. *Comm. ACM*, 44(5), May 2001.
- [2] Cynthia Bailey Lee, Yael Schwartzman, Jennifer Hardy, and Allan Snaveley. Are user runtime estimates inherently inaccurate? In Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 3277 of *Lecture Notes in Computer Science*, pages 253–263. Springer Berlin / Heidelberg, 2005.
- [3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37:164–177, October 2003.
- [4] Dr. F. L. Bauer. Software engineering. In Brian Randell Peter Naur, editor, *Software engineering: Report of a conference sponsored by the NATO Science Committee*, Alemania, 1968.
- [5] P. Bernstein. Middleware: a model for distributed system services. *Communications of the ACM*, 39(2), February 1996.
- [6] T. Bishop and R.K. Karne. A survey of middleware. In *18th International Conference on Computers and Their Applications*, Honolulu, Hawaii, March 2003.
- [7] Robert G. Brown. *Engineering a Beowulf-Style Compute Cluster*. Duke University Physics Department, 2006.
- [8] Marti Campoy, A. Perles Ivars, and J. V. Busquets Mataix. Static use of locking caches in multitask preemptive real-time systems. In *In Proceedings of IEEE/IEE Real-Time Embedded Systems Workshop (Satellite of the IEEE Real-Time Systems Symposium)*, 2001.
- [9] Junwei Cao, Daniel P. Spooner, Stephen A. Jarvis, and Graham R. Nudd. Grid load balancing using intelligent agents. *FUTURE GENERATION COMPUTER SYSTEMS*, 21(1):135–149, 2005.
- [10] Christopher Clark, Keir Fraser, Steven H, Jakob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of

- virtual machines. In *In Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 273–286, 2005.
- [11] Gionatan Danti. Kvm i/o slowness on rhel 6. www.ilsistemista.net/index.php/virtualization/11-kvm-io-slowness-on-rhel-6.html, March 2011.
 - [12] Benoît des Ligneris, Stephen Scott, Thomas Naughton, and Neil Gorsuch. Open source cluster application resources (oscar): design, implementation and interest for the [computer] scientific community, 2003.
 - [13] M. Dobber, G. Koole, and R. van der Mei. Dynamic load balancing experiments in a grid. In *Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 2 - Volume 02*, CCGRID '05, pages 1063–1070, Washington, DC, USA, 2005. IEEE Computer Society.
 - [14] Menno Dobber, Ger Koole, and Rob Van Der Mei. Dynamic load balancing for a grid application. In *In Proceedings of HiPC 2004*, pages 342–352. Springer-Verslag, 2004.
 - [15] J. Duell, P. Hargrove, and E. Roman. The design and implementation of berkeley lab’s linux checkpoint/restart. Technical report, Berkeley Lab, 2003.
 - [16] Douglas Eadline. The true cost of hpc cluster ownership. www.clustermonkey.net/content/view/262/1/1/0/, July 2009.
 - [17] Dror G. Feitelson and Ahuva Mu’alem Weil. Utilization and predictability in scheduling the ibm sp2 with backfilling. In *In Proceedings of the 1st Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (IPPS/SPDP-98)*, pages 542–547, Los Alamitos. IEEE Computer Society, 1998.
 - [18] Hanhua Feng, Vishal Misra, and Dan Rubenstein. Pbs: A unified priority-based cpu scheduler. Technical report, In Proc. of ACM Sigmetrics, 2007.
 - [19] C.J. Fidge. Real-time schedulability tests for preemptive multitasking, 1996.
 - [20] Ada Gavrilovska, Sanjay Kumar, Himanshu Raj, Karsten Schwan, Vishakha Gupta, Ripal Nathuji, Radhika Niranjana, Adit Ranadive, and Purav Saraiya. High-performance hypervisor architectures: Virtualization in hpc systems, 2007.
 - [21] Richard Gibbons. A historical application profiler for use by parallel schedulers. In *Proceedings of the Job Scheduling Strategies for Parallel Processing*, IPPS '97, pages 58–77, London, UK, 1997. Springer-Verlag.

- [22] Anastasios Gounaris, Jim Smith, Norman W. Paton, Rizos Sakellariou, Alvaro A. A. Fern, and Paul Watson. Adapting to changing resource performance in grid query processing. In *In Data Management in Grids, First VLDB Workshop, DMG 2005*, pages 30–44. Springer, 2005.
- [23] R. Haupt. A survey of priority rule-based scheduling. In Springer, editor, *OR Spectrum*, volume 3834 of *Lecture Notes in Computer Science*. Springer, 1989.
- [24] Matthias Hovestadt, Odej Kao, Axel Keller, and Achim Streit. Scheduling in hpc resource management systems: Queuing vs. planning. In *Proc. of the 9th Workshop on Job Scheduling Strategies for Parallel Processing*, pages 1–20. Springer, 2003.
- [25] Wei Huang, Matthew J. Koop, Qi Gao, and Dhabaleswar K. Panda. Virtual machine aware communication libraries for high performance computing. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing, SC '07*, pages 9:1–9:12, New York, NY, USA, 2007. ACM.
- [26] Wei Huang, Matthew J. Koop, and Dhabaleswar K. Panda. Efficient one-copy mpi shared memory communication in virtual machines. pages 107–115, 2008.
- [27] Joshua Hursey, Jeffrey M. Squyres, Timothy I. Mattox, and Andrew Lumsdaine. The design and implementation of checkpoint/restart process fault tolerance for open mpi. In *In Workshop on Dependable Parallel, Distributed and Network-Centric Systems(DPDNS), in conjunction with IPDPS*, March 2007.
- [28] Cluster Resources Inc. Moab cluster suite. www.clusterresources.com/products/moab-cluster-suite.php.
- [29] S.A. Jarvis, D.P. Spooner, H.N. Lim Choi Keung, G.R. Nudd, and J. Cao. Performance prediction and its use in parallel and distributed computing systems. In *PROCEEDINGS OF THE 17TH INTERNATIONAL SYMPOSIUM ON PARALLEL AND DISTRIBUTED PROCESSING (2003)*. *IEEE COMPUTER SOCIETY*, page 2006, 2006.
- [30] Morris A. Jette, Andy B. Yoo, and Mark Grondona. Slurm: Simple linux utility for resource management. In *In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003*, pages 44–60. Springer-Verlag, 2002.
- [31] Katarzyna Keahey and Tim Freeman. Contextualization: Providing one-click virtual clusters. In *Proceedings of the 2008 Fourth IEEE International Conference on eScience*, pages 301–308, Washington, DC, USA, 2008. IEEE Computer Society.

- [32] Gaurav Khanna, Umit Catalyurek, Tahsin Kurc, P. Sadayappan, and Joel Saltz. A data locality aware online scheduling approach for i/o-intensive jobs with file sharing. In *Proceedings of the 12th international conference on Job scheduling strategies for parallel processing*, JSSPP'06, pages 141–160, Berlin, Heidelberg, 2007. Springer-Verlag.
- [33] Dalibor Klusáček. *Event-based Optimization of Schedules for Grid Jobs*. PhD thesis, Masaryk University, Brno, Czech Republic, 2011.
- [34] Aleksandar Lazarevic. *Probabilistic grid scheduling based on job statistics and monitoring information*. PhD thesis, University College London, London, England.
- [35] Hui Li, David Groep, and Lex Wolters. Workload characteristics of a multi-cluster supercomputer. In Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 3277 of *Lecture Notes in Computer Science*, pages 33–53. Springer Berlin / Heidelberg, 2005.
- [36] David A. Lifka. The anl/ibm sp scheduling system. In *In Job Scheduling Strategies for Parallel Processing*, pages 295–303. Springer-Verlag, 1995.
- [37] Jiuxing Liu, Wei Huang, Bulent Abali, and Dhabaleswar K. Panda. High performance vmm-bypass i/o in virtual machines. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, Berkeley, CA, USA, 2006. USENIX Association.
- [38] Michael R. Marty. *Cache coherence techniques for multicore processors*. PhD thesis, Madison, WI, USA, 2008. AAI3314233.
- [39] Marvin McNett, Diwaker Gupta, Amin Vahdat, and Geoffrey M. Voelker. Usher: An extensible framework for managing clusters of virtual machines. In *Proceedings of the 21st Large Installation System Administration Conference (LISA)*, November 2007.
- [40] Mark F. Mergen, Volkmar Uhlig, Orran Krieger, and Jimi Xenidis. Virtualization for high-performance computing. *SIGOPS Oper. Syst. Rev*, 40:8–11, 2006.
- [41] Tran Ngoc Minh and Lex Wolters. Using historical data to predict application runtimes on backfilling parallel systems. In *Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, PDP '10, pages 246–252, Washington, DC, USA, 2010. IEEE Computer Society.
- [42] Ahuva W. Mu'alem and Dror G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. *IEEE Trans. Parallel and Distributed Systems*, 12(6):529–543, 2001.

- [43] Susanta Nanda and Tzi cker Chiueh. A survey of virtualization technologies. Technical report, 2005.
- [44] Gregory F. Pfister. *In Search of Clusters: The Ongoing Battle in Lowly Parallel Computing*. Prentice Hall, 2 edition, 1998.
- [45] Xiao Qin. Design and analysis of a load balancing strategy in data grids. *Future Gener. Comput. Syst.*, 23:132–137, January 2007.
- [46] David Jackson Quinn, David Jackson, Quinn Snell, and Mark Clement. Core algorithms of the maui scheduler. pages 87–102, 2001.
- [47] Joseph D. Sloan. *High Performance Linux Clusters: With OSCAR, Rocks, openMosix, and MPI (Nutshell Handbooks)*. O’Reilly Media, Inc., 2004.
- [48] Srividya Srinivasan, Rajkumar Kettimuthu, Vijay Subramani, and P. Sadayappan. Selective reservation strategies for backfill job scheduling. In *Revised Papers from the 8th International Workshop on Job Scheduling Strategies for Parallel Processing, JSSPP ’02*, pages 55–71, London, UK, UK, 2002. Springer-Verlag.
- [49] M. Stillwell, F. Vivien, and H. Casanova. Dynamic fractional resource scheduling vs. batch scheduling. 2011.
- [50] Mark Stillwell, David Schanzenbach, Frederic Vivien, and Henri Casanova. Resource allocation using virtual clusters. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, CCGRID ’09*, pages 260–267, Washington, DC, USA, 2009. IEEE Computer Society.
- [51] Dan Tsafir, Yoav Etsion, and Dror G. Feitelson. Modeling user runtime estimates. In *In 11th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP 2005)*, pages 1–35. Springer-Verlag, 2005.
- [52] Dan Tsafir, Yoav Etsion, and Dror G. Feitelson. Backfilling using system-generated predictions rather than user runtime estimates. *In IEEE TPDS*, 18:789–803, 2007.
- [53] Deepti Vyas and Jaspal Subhlok. Volunteer computing on clusters. In Eitan Frachtenberg and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 4376 of *Lecture Notes in Computer Science*, pages 161–175. Springer Berlin / Heidelberg, 2007.
- [54] Yonghong Yan and Barbara Chapman. Comparative study of distributed resource management systems – sge, lsf, pbs pro, and loadleveler.
- [55] Kenneth Yoshimoto. Catalina. www.sdsc.edu/catalina/.

- [56] Lamia Youseff, Rich Wolski, Brent Gorda, and Chandra Krintz. Paravirtualization for hpc systems. In *Proceedings of the 2006 international conference on Frontiers of High Performance Computing and Networking*, ISPA'06, pages 474–486, Berlin, Heidelberg, 2006. Springer-Verlag.
- [57] Chi Zhang, Xin Yuan, and A. Srinivasan. Processor affinity and mpi performance on smp-cmp clusters. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1 –8, april 2010.

Appendix A

Middleware

We have mentioned a little bit about certain applications that run in the slave nodes and others that run in the server, however we haven't said much about them besides the fact that they collect information logs, monitor nodes and send jobs. Here we explain a little more about how this pieces of software work and how "middleware" can mean almost anything.

A.1 Definition

It is believed that the term middleware was used for the first time in the NATO conference back in 1968 [4] where figure A.1 was presented, in this case, the application software had to communicate with the file system, however each vendor programmed the routines for accessing it in a different way and with different calls, thereof without the middleware, each time that the application was to be executed in another system it was necessary to rewrite the whole module that used the hard drive services, however with the middleware only compatibility with the new Operating System was needed in it and the software itself didn't need to be changed.

After NATO 1968, the term middleware has been used to describe many and distinct software products [6], so with the mere mention of middleware is not possible to know immediately what is meant without entering first in the appropriate context. In contrast whit what happens with terms like "kernel", "console", "GUI", etc. In this case the term kernel, even though it can be used to define other things, it's most common meaning is "operating system kernel" unless otherwise specified.

The term middleware has been used to define a software layer between the OS or the network and some application in order to allow the communication between two programs [5]. Also it has been defined a as integration tool to connect software modules [1].

Bishop and Karne propose the following definition for middleware: "Middleware is the software that assists an application to interact or communicate

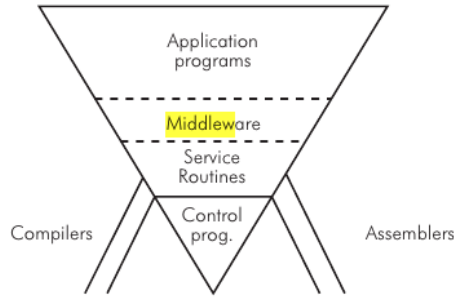


Figure A.1: In 1968, the middleware was used as an intermediate layer between applications and heterogeneous services that could change.

with other applications, networks, hardware, and/or operating systems. This software assists programmers by relieving them of complex connections needed in a distributed system. It provides tools for improving quality of service (QoS), security, message passing, directory services, file services, etc. that can be invisible to the user.”

Lets notice that this definition tries to cover as many software products as possible, so it's actually a little bloated for most of the cases because the middleware typically targets an specific problem on a particular level; this, however doesn't affects our purpose that is to define the middleware for computing cluster which we do in A.3.

A.2 Classification

There exists different ways to classify the middleware, it all depends on the characteristics considered more or less important, for this text the classification of Bishop was adopted, that is shown in figure A.2, this divides the the middleware in two main branches: integration and application. Here we give an overview of each classification, for more information see [6].

The branch of **integration** refers to the products that facilitate the communication between applications and services or servers, for example the service Remote Procedure Call (RPC) allows to request in a standardized way an particular service, it is even possible to request a particular version of the service to a server through a particular port without having to care about which port the service is actually using because the appropriate port is returned too (RPC can controll TFTP, FT, NFS, etc.).

Other example is in the case of the agents where each one can take decisions in the name of the user, this is used by some internet advertising services where each agent gathers information about the search items of the user, and based on them offers other products or services that could be of interest for them.

It is possible to offer components that perform an specific task, for example there exists some grid services that calculate the Fast Fourier Transform (FFT),

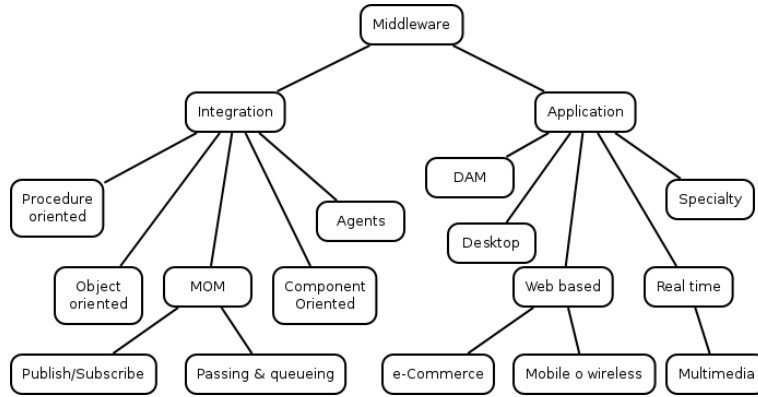


Figure A.2: Middleware classification depending on it's purpose.

in this case the middleware takes care of managing the connection to the service, send the data that will be used to perform the transform and retrieve the results; taking care of hiding the all the obscure details about secure connection, service lookup, etc.

Other kind of middleware that has turned popular is the Simple Object Access Protocol (SOAP), this facilitates the communication between object oriented programming languages of different families. What SOAP does is to transform the objects that are to be sent into a platform and programming language independent format that is sent through some network protocol between the applications.

The branch of **applications** refers mainly to the products that offer well defined functionality specific to make it easier the development of applications that require a particular set of services in a way such that the required functionality can be “embedded” into the new application.

There exists web services that monitor the search terms of the user and the pages and offer products and services depending on what seems to be of interest to the user, they also show advertising based on the OS that the user is running, etc. This services have to be platform agnostic and provide a good set of actions at the same time, for example to perform a secure connection to the user's bank account in order to realize payments, etc.

The real time middlewares are specially concerned about attending the queries of the users as soon as they are sent, otherwise the data requested could not be useful anymore, an example of this are the Massive Multiplayer Online Games (MMOG), this in real time take the actions performed by the user, send them through internet, perform some computation and forward it to every player sharing the session, in this cases, getting something that occurred two minutes before in the game is plain useless.

There exists also desktop applications with similar functionality, some of them show in the screen updates of news channels or any other content that could be of interest like traffic, weather and others.

There is still much more that we could say about middleware, however that's not the objective of this work to disclose every detail of each of this softwares as variate as useful, so from now on, we concentrate on a particular kind of middleware that is use for computing clusters.

A.3 Middleware parar clusters

As we just see previously, there are lots of kinds of middleware, in this section we try to characterize one particular type that is used for computing clusters, both commercial and free software. We cover in particular some popular products like TorquePBS, Oracle Solaris Cluster, SULRM and OSCAR without actually entering deep into the details of each one of them, as they are good or weak in different ways.

A.3.1 Randomly available resources

Lets suppose that an user has a set of n jobs that would like to execute and that the company has a set of m computers available to execute such jobs, some of the m machines are completely idle while others are executing jobs from other users. The user faces the dilemma of deciding which machines pick to send the jobs to; because selecting idle computers will allow the jobs to finish faster, it is natural for the user to want to know which are being used and which are not.

It is not viable for users or big groups to be everyone on their own polling the nodes in order to check which are less busy at the time in order to run their jobs, neither is viable to ask everyone no to use the nodes because they are needed or manually schedule the time that everyone can use them, because it's possible that some will have more time that the one they actually need and others will be short of time.

Part of the job of the middleware for computing clusters is precisely to maintain a control of how many and which nodes (or resources) have been requested by each user in order to assign automatically the resources that each users needs, this is made by providing means to request and liberate resources. The user usually requests a number of resources, and resources are assigned from the pool of free resources.

Even though it is better to let the middleware decide which resources to assign, sometimes the user would want to indicate a set of nodes to use to run the application, many of this middlewares allow the user to explicitly indicate the hostname of the nodes that want to use, or name capabilities or attributes.

A.3.2 The problem with the access

There is also another problem while trying to execute parallel program that is to be divided in processes in tens of computers or sessions, it wouldn't be feasible to expect the user to individually login into each of the machines by entering user name and password for each session to be launched.

Nevertheless it's not feasible either to allow the access without password because it's possible that the user will leave in some nodes confidential information about their work or research and such don't want any other user to be able to access it.

When one has hundreds of nodes it's also a lot of work for the administrator to take the task of administering the users in each node, assign passwords, etc. so a efficient way of managing users is needed, or even better, if it wasn't needed to administer the users at all and even so, allow the users to execute their jobs and keep the privacy.

The middlewares for clusters are compatible with some popular authentication mechanisms depending on the policies and resources available, for example it's possible to use mechanisms of remote sessions like `rsh`, secure connection with `ssh`, secure connection without password authentication with the combination of `ssh` and `rsa`, authentication daemons like MUNG and services like Kerberos, it's possible to choose which to use or migrate from one to another just by tweaking some configuration parameters.

A.3.3 User's shared directory

Generally, when users are created in a computer, they are also created a personal directory where they can store their working files. Even if it's not strictly necessary that the user have a personal directory, due to security and privacy concerns it's convenient for it to be that way.

Now lets imagine that the user requests the execution of a program and it allowed the middleware to decide where it should be executed. Suppose that the output of the program is a file that was stored in the user's personal folder in the machine where it executed, nevertheless the user doesn't know in which node the program executed, so a way of entering to each computer and search for the output file would be needed, or include within the program a route that sends a mail with the name of the machine used or send the file itself.

Both of the described solutions have their problems, to begin with, there is a lot of time wasted by the user to search for the file that could be used for something else, and other problem is that it could compromise the portability of the code.

The resource managers also allow the easy integration with dynamic directory services like the Network File System (NFS) that comes by default with most of the *nix systems, this enables the sharing of a group of directories between many computers. NFS is an excellent service for small clusters, but maintaining the connections of many nodes represents a good challenge for the bandwidth.

There are better solutions that also work like the Lightweight Directory Access Protocol (LDAP) that dynamically mounts a remote directory when it's needed, it leaves it like that for some time and if it's not used during that time then closes the connection; this is a useful solution specially because it's not very sensible to the changes.

In this way the user only sees one directory, the same, in every node of the cluster, so there is no worrying about where the file was stored, with this kind of solutions is also possible to use diskless nodes.

A.3.4 Add and remove resources

Some of the recurrent problems that a system administrator faces is to add new resources, give maintenance, fix and drop some of the existing resources that are available. For instance, one of the new generation computers could do the work that before was done by four of the five year old nodes.

I wouldn't be feasible to ask each of the users to stop using a group of nodes that are to be deprecated or given maintenance and then wait for the to confirm that their jobs finishes executing, specially because it's possible that they don't even read their mail too often.

Middlewares allow to specify which nodes are available to execute jobs and which are to be "drained" from jobs, if a node is to be replaced, it's possible to tell the resource manager to stop sending jobs to that particular node or nodes an even to send an email to the system administrator when that node is free and so can be replaced.

In the same way it's not necessary to notify each of the users when a new node is added, unless it has some special hardware that could be of interest for some, it's enough to tell the resource manager that such node is available for executing jobs and all the users will benefit from the new addition.

A.3.5 Statistics

Naturally, and specially when the resources are being rented to some users or companies, the people in charge of the computing cluster is interested in obtaining information about what is happening with each of the nodes.

They could be interested in knowing, for instance, the average waiting time of the jobs, if they are too high it would probably be convenient for them to get more resources in order to give a better service to the users, or to revise the job policies implanted, like maximum number of nodes or upper limit for the run time request.

They would also be interested in knowing what percentage of the computers are used a day. If the wait times are too high and some computers are not used at all most of the time, probably there is some problem with the job policies, and tweaking them could improve the service.

The middleware collects lots of information and also has utilities to format it in a way that is easy to read for everyone, specially for the responsible of the cluster, allowing to take decisions more easily.

Appendix B

SLURM

Because we are making use of the Simple Linux Utility for Resource Management (SLURM) to implement the solution proposed in this work, through the text we have mentioned about this Resource Manager (RM), in this appendix we give a more detailed study of its architecture to show yet another reason why we decided to use it.

We start by studying the life cycle of a typical job in the computing cluster and see the information that is available or can provide about the jobs, also the information available at the end of its execution. Lastly we see the facilities available to modify or tweak to our needs the behavior of SLURM through the use of plugins.

B.1 Job execution

The process of executing jobs in a cluster can be as simple or as complex as needed depending on the user's necessities, we can search manually for the nodes that are available, request them, request the amount of time, etc. or we can allow the RM to do it all for us.

In figure B.1 we can see a basic (generic) diagram of the life cycle of the jobs in a cluster from the point of view of the RM: the users start by submitting their jobs, the middleware (for a description of middleware see appendix A) queues them, the scheduler (internal or external) orders the queue in base of some rules, the middleware sends the “next job in the queue” to the set of machines assigned to run them, the nodes execute the jobs and notify the RM about their exit status.

First of, the user's program has to be in a place where all the nodes can find it, depending on the kind of job, it's possible that it's installed in all the nodes, or that it's copied to a file server or that it's compiled from the nodes before executing it.

If all that applies, the set of nodes and their status can be found as follows:

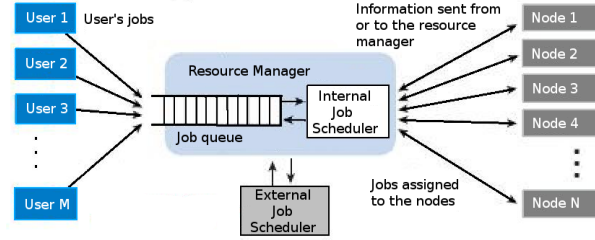


Figure B.1: Diagrama de ejecución de un trabajo

```
farfan@lamia:~$ sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
particio*   up    infinite     5  alloc lamia[2-3]
particio*   up    infinite     5  idle  lamia[4-6]
particio*   up    infinite     1  down* lamia
```

In this particular case, we can see a total of six nodes, from which two were busy (alloc), one was unavailable (down) and three were available for running jobs (idle).

We can do more in depth inspection to check the characteristics of the nodes that are available as follows (only part of the output is shown):

```
farfan@lamia:~$ scontrol show node --oneline
  NodeName=lamia4 Arch=i686 CoresPerSocket=1 CPUAlloc=0
CPUErr=0 CPUTot=2 Features=(null) Gres=(null) OS=Linux RealMemo
ry=493 Sockets=1 State=IDLE ThreadsPerCore=2 TmpDisk=11264
Weight=1 BootTime=2011-05-26T12:21:46 SlurmdStartTime=2011
-05-26T13:19:49 Reason=(null)
  NodeName=lamia5 Arch=i686 CoresPerSocket=1 CPUAlloc=0
CPUErr=0 CPUTot=1 Features=(null) Gres=(null) OS=Linux RealMemo
ry=755 Sockets=1 State=IDLE ThreadsPerCore=1 TmpDisk=11264
Weight=1 BootTime=2011-05-26T12:21:59 SlurmdStartTime=2011
-05-26T13:19:49 Reason=(null)
  NodeName=lamia6 Arch=i686 CoresPerSocket=1 CPUAlloc=0
CPUErr=0 CPUTot=1 Features=(null) Gres=(null) OS=Linux RealMemo
ry=247 Sockets=1 State=IDLE ThreadsPerCore=1 TmpDisk=11264
Weight=1 BootTime=2011-05-26T12:22:01 SlurmdStartTime=2011
-05-26T13:19:49 Reason=(null)
```

As we can see, the machine 4 has 2 cores and the rest have 1, giving a total of 4 cores that can be requested to execute a given application, even more, all the machines are i686 compatible.

After having verified the nodes that we are interested on using, we can request the execution of a process in the desired nodes this way:

```
farfan@lamia:~$ srun -N3 -t5 -w lamia[4-6] hostname
lamia5
lamia4
lamia6
```

This will block the terminal until the three (parameter `-N3`) resources required (lamia 4 through 6) are available at the same time for five minutes (parameter `-t5`) to execute the application, the current terminal is used as standard input, output and error.

Unless there are very good reasons for doing it that way, the average user will only be interested in knowing the maximum number of nodes and the greater amount of time that can be requested

```
farfan@lamia:~$ scontrol show partition
PartitionName=partition
  AllocNodes=ALL AllowGroups=ALL Default=YES
  DefaultTime=NONE DisableRootJobs=NO Hidden=NO
  MaxNodes=UNLIMITED MaxTime=UNLIMITED MinNodes=1
  Nodes=lamia,lamia[2-6]
  Priority=1 RootOnly=NO Shared=NO PreemptMode=OFF
  State=UP TotalCPUs=9 TotalNodes=6
```

The average user will let the RM allocate the first subset of nodes that meets the requirements of the job, this is because, generally, it's faster to wait for any subset to get free than to wait for a particular subset to get free, this is achieved simply by omitting the parameter "`-w <nodes>`."

For non interactive processes that require, though, to realize some preparing or cleaning tasks, it's possible to create a bash script with the parameters to be sent to the RM along with the job:

```
#!/bin/bash
#SBATCH --nodes=5
#SBATCH --time=10
#SBATCH --job-name=CalculaPi
#SBATCH -o pi.txt
mpicxx PiM.cpp -o /tmp/PiM
mpirun /tmp/PiM 90000
rm -f /tmp/PiM
```

In this example any set of five nodes is being requested for ten minutes, the name of the job is `CaluclPi` and the standard output is to be saved in a file named `pi.txt` of the file and directory where the job runs. Lets notice that

here we are compiling the program before executing it and we delete it after it finishes, this are the preparing and cleaning tasks for this job.

If that bash script is named `Pi.sh`, we can queue it executing the following command: `sbatch Pi.sh`. This script is sent to every one of the nodes where they first compile the program and will save the binary in `/tmp/PiM` (lets remember that its in a shared directory), after that it is executed using MPI and at the end the binary is deleted. The middleware is responsible of redirecting the standard output to the specified file.

B.2 Queuing process

At the time of writing this document there was not enough documentation about the path followed by a job that is to be executed in the cluster from the time when it's submitted, nevertheless here we try to describe such path to the best of our knowledge in order to show that just knowing a little bit about SLURM we can implement even an scheduler.

It starts by requesting computing resources to allocate the job, this is done with the commands shown in section B.1. What this does is to fill a data structure with information that defines the requirements of the job and then uses a Remote Procedure Call (RPC) to the `slurmctld` daemon, this daemon in turn validates the requirements and notifies whether or not is possible to execute the job immediately, if it was put in the waiting queue or if there was an error.

The process of resource selection requires various stages, some of the which are done with the help of plugins.

1. Calls the presentation plugin `job_submit` to modify the request in an appropriate manner.
2. Checks that everything is in order with the job (queue accessible for the user, limits, etc.)
3. If it's a high priority job, allocate the resources immediately, otherwise, validate that it could be allocated if there were no other jobs running.
4. Establish which nodes could be used to run the job.
5. Call the selection plugin `select` to chose the best resources for the request.
6. The selection plugin will consider the topology to chose the resources if it's available.

To allocate a job, the command `srun` is always used, this sends the RPC request to the `slurmctld` daemon using the parameters from the command line and those in environment variables. The logic used for the resource selection is withing the `slurmctld` daemon. If for any reason it's not possible to allocate the resources, the appropriate notification is sent and `srun` tries again at some other

time. Once the resources are allocated, a *certificate* that identifies the resources that can be used is generated, this certificate is validated by the nodes before executing the jobs.

With a single allocation it's possible to execute many tasks (known as steps), so `slurmstepd` is called for each step to be executed, this program among other things does the following:

- Configure the Message Passing Interface (MPI) through some plugin (the way of initializing is different for each vendor).
- Calls the `switch` plugin to configure the network.
- Generates a container for each task using the `proctrack` plugin.
- Changes the User IDentificator (UID) for that of the appropriate user (otherwise it would be executed as administrator).
- Configures the Input/Output for the task using either files or sockets with the `srun` command.
- Configures the environment variables.
- Calls `fork` or `exec` for the tasks.

There are various ways in which a *step* of a task can finish, the simplest one is when the process finishes successfully. `srun` realizes that the job finished and notifies the `slurmctld` daemon about the event. `slurmctld` only registers the termination. Others are that the task *step* finished because it was canceled, the time ran out, etc. what happens is described in the following paragraph.

A job can terminate due to the user (job canceled) or because of the system. The termination requires that the `slurmctld` daemon notifies the `slurmd` daemons in every node that the job is to be terminated. After that the `slurmd` daemon does the following:

1. Sends the signals `SIGCONT`¹ and `SIGTERM`² to the user's tasks.
2. Waits `KillWait` seconds for it to finish.
3. Send the signal `SIGKILL`³ to all the tasks.
4. Wait for the tasks to finish.
5. Execute some finish program (if configured).
6. Send `epilog_complete` to `slurmctld` through RPC.

¹Resume execution

²Finish the execution cleanly

³Kill the process at OS level

B.3 Job scheduling

As we saw in section B.1, SLURM is conveniently divided in modules, daemons and plugins with well defined purposes. This makes it specially attractive in case that it's needed to write a personalized scheduler or influence in the existing one.

There are mainly two ways in which we can influence the order in which the jobs will be executed through the use of a planification plugin, one is to put an initial priority of zero that will prevent them from executing at all until the scheduler (extern or wiki) explicitly changes their priority.

The other is using some kind of backfill that periodically alters the priority of the jobs, this changes their order in the queue. For example, increasing the priority of a job may make it start sooner, though it could also move the start time of the other jobs too.

A plugin of type `wiki` has the advantage that the underlying scheduler is generic in a way such that can be used with more than one middleware, like in the case of Maui and Moab, both work with a variety of RM without the need of modifications for either one.

An important aspect of the wiki plugin is that it allows the developer to have a complete control of each of the planning phases, this is not necessarily good or bad unless aspects like development time, middleware dependencies, etc. are taken into account. For the kind of work proposed in this text, unfortunately, there is not enough time for a solution of this magnitude.

Both wiki and backfill have to follow the same Application Programming Interface (API), the difference is that wiki allows an external agent to do the planning or tell it what to do.

Appendix C

Scheduling and planning

In this appendix we study the three different levels of planning that is convenient to distinguish in order to have a better understanding of the scope of this work, after that we see some categories of scheduling algorithms for clusters and some of the representatives of each category.

C.1 Planification levels

We haven't found in the literature what we call here "planification levels", maybe because generally it's implicit in the level in which each job is being done. Nevertheless, with the purpose of limiting the scope of this work, here we define three levels of planning that we can distinguish: grid, cluster and application.

In any of the levels we are planning the work that is pending to do while taking into account the work that is being done along with the resources allocated and free, with this, one of the objectives of the planification for any of the levels is the efficient use of the resources, keeping them busy as long as possible.

Grid (figure C.1): In this planification level, the resources for the jobs of the users will typically be ¹ **clusters** or **supercomputers**, the middleware of the grid, thereof, forwards the jobs to the clusters that it considers appropriate for executing it.

Because this level is composed by many heterogeneous elements, many research works have been made with the purpose of exploiting the dynamic nature of its individual resources [9, 22, 29, 34], we are not very interested in this planification level, so no further mention is done.

Cluster (figure C.2a): In this level the work to schedule are also the user's

¹In some kinds of grids, for example, Globus, the users can allow their personal computers to be used to execute jobs when they are idle for a long time (instead of running a silly screensaver).

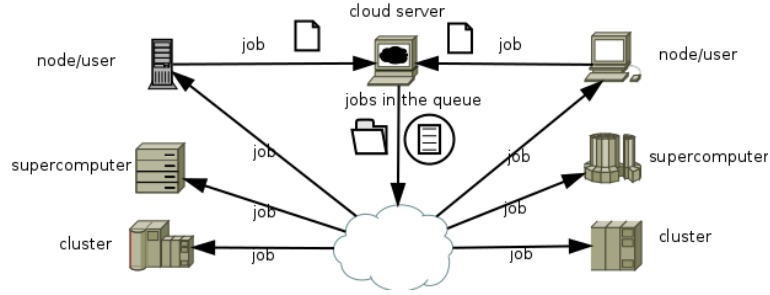


Figure C.1: Planification at cloud/grid level: one server and many independent and different resources.

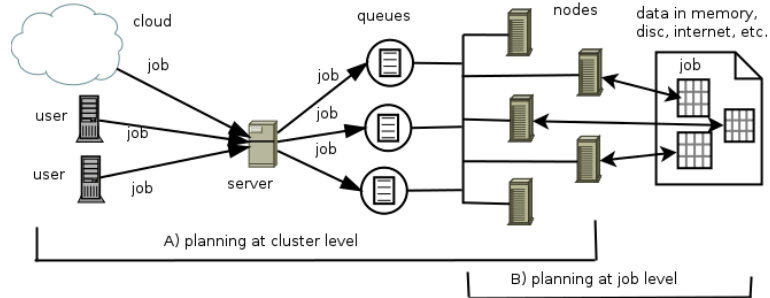


Figure C.2: A) The server distributes the job between the resources. B) The process distributes the data among the allocated nodes.

jobs² while the resources can be, depending of the configuration, nodes (computers), processors, processor cores [57] and / or hardware threads³.

Application (figure C.2b): This is the user's job, here it's resources can be complete nodes⁴ or cores⁵. What is planned here, instead, is the total amount of work to do, for example, elements of a matrix, database objects, files [32], etc.

The job of the cluster schedulers, then, finishes once it allocates a subset of resources to a job and it starts its execution, as we saw in appendix B, the only thing that the middleware sees is a plain text file with the requirements of the job and the command line commands that starts it, for this reason, the cluster scheduler cannot plan also the job that the application needs to finish.

²That can be considered external if they come from the grid or internal if a user accessed directly to the cluster to request the execution.

³Also can be scheduled the memory, I/O [45], licenses, etc. For this work, however, the unit are nodes.

⁴Specially when threads are used.

⁵Mainly for MPI jobs where one process is launched per each core.

The fact that there are many publications about grid load balancing, like [13,14] just to mention some, could confuse those relatively new to the subject, not being obvious that before making their own planning, the job already passed through the scheduler of the grid and cluster.

C.2 Kinds of cluster schedulers

Hovestadt et al. [24] identifies two main ways to do job schedule in clusters and grids depending on whether the plan is complete or partial or takes into account only free resources or also allocated ones.

The first are the **queue based systems**, this where the first to be developed as they are simpler. It counts with many queues, each one with different limits in the number or kinds of resources requested, the time limit, etc. Also the queues can be activated or disabled based in the hour or date ⁶.

The job of a queue system is to assign free resources to pending jobs. The request with the highest priority always is the head of the queue, it's possible to assign resources to more than one head at the same time and additionally it's possible to use criteria like the priority of the queue and the best fit to select between the pending jobs.

In the event that there are not enough resources available to start executing the head job from any queue, the system waits until some resources get freed, Nevertheless, the resources available at that moment can still be used through backfilling policies.

Because the head of the queue is always the one to try to execute, this systems doesn't really require information about the expected execution time of the jobs, unless backfilling is desired.

One of the problems with this kind of systems is that it's difficult to tell when a new job will start running ⁷, nevertheless the "scheduling cost" ⁸ is relatively low and executing the next job is fast.

The other ones are the **systems based on a schedule**, this make the present and future execution plan, this gives as result that it's now possible to know the start and finish date for each job, obviously for this to work the runtime estimates are needed, this is also important because this way it's possible to make reservations ⁹.

Every time a new request is sent or a job finishes before it's expected, a new plan is made. Everything except reservations is deleted, the queue is sorted depending on the scheduling politics and then they are assigned the earliest start time possible. This process is called rescheduling and is only necessary for some algorithms.

⁶For example it's possible to create queues that can only be used during weekends.

⁷Because the queue can be sorted every time that a new job is sent or one is canceled.

⁸Time that it takes to sort the queue.

⁹If a single grid job can be executed in many clusters at once then it's expected to start running at the same time in all of them.

Making the rescheduling implies some kind of backfilling, because every job is schedule to start as soon as possible, some can be planned to start even before others that where already scheduled as long as their reservations are not affected (moved) by doing so.

One of the disadvantages of the schedule based algorithms is that they are more complex computationally than the queue based, with today's processors this is not a big problem though, other is in case that the users can peek into the schedule, they could get mad because some jobs are planned to start before than their own.

C.3 Scheduling algorithms

Now let's see some examples of scheduling algorithms and how they work, we describe very quickly how some of them could be implemented together with some properties of the resulting schedule and the way they select the jobs from the defined queues.

Many planing algorithms have been proposed, nevertheless not all of them are used in practice, for instance those that do data mining through the job registry [21], even though they seem to work well, the latency introduced could make them impractical or hard to develop.

The **queue based** ones give a priority to each of the jobs in the queue and only select the one with the greater priority (the head of the queue) to be next one executed, the criteria can be of any kind like if the user is paying or not, type of job, special requirements, etc.

First Come First Served (FCFS) maybe the simplest of all, don't even sort the queue, it only takes the one in the head every time and run them in the same order they arrived, if at some point there are not enough resources for the next one to run, it simply waits for some to get freed until the job can run, its main disadvantage is that many resources stay idle for randomly long periods of time.

Shortest Job First (SJF) is similar to FCFS with the difference that it first sorts the queue based on the size (length and width) of the jobs giving the highest priority to the smallest of them. At first it will assign lots of jobs, but soon after there will be only big jobs in the queue that are more difficult to schedule efficiently.

Longest Job First (LJF) it's quite the opposite to SJF, however it could be more useful, for instance if the job is very small it could be executed in the personal computers of the user instead of on the cluster, also using backfilling the smaller jobs could be executed anyway in order to keep the machines busy until there are resources available for the bigger jobs.

Earliest Deadline First (EDF) if the user can set a deadline to run the job such as "if it doesn't run by some date don't run it at all" this algorithm can be used as it will try to execute as many as possible based on their deadlines.

The **schedule based** ones make the plan for all the jobs in the queues at once, this way it's possible to know in advance the time at which (most likely)

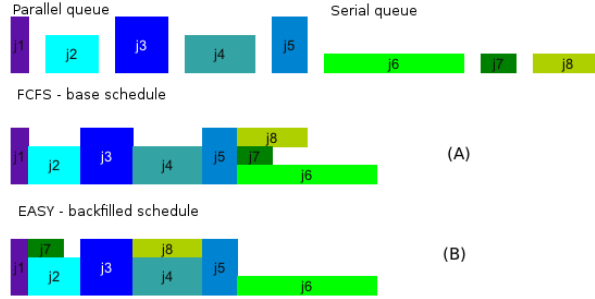


Figure C.3: A) FCFS: the jobs are assigned in the order they enter the queue. B) Backfill: if there is a hole and a job of its size, it is allowed to run.

a particular job will start running and even which nodes it will use. The queue based algorithms can be ported to behave this way very easily.

As we already mentioned the schedule could be modified by some events like the arrival of a job to the queue, the cancellation of one or when one of them finishes before it was expected to (which is very common [2]).

Better Gap (BG) searches for the gaps in the schedule, for every job in the queue it tries to fit in in the gaps big enough for them to fit it, some criteria must be followed to decide which of the gaps to use, for example to increment the performance of the algorithm the first gap big enough could be used.

Some ways of enhancing the performance in the algorithms and increasing the utilization of the cluster at the same time have been proposed depending on the particular necessities of the people behind the idea or the type of jobs that run in the cluster in order to take advantage of them.

Probably the most successful enhancement is the **backfilling** because it works well with any other algorithm. As its name suggests, after having created the schedule in the usual way, this algorithm searches for holes in the schedule and also for jobs that can fit in them in order to keep the resources busy. There are two main ways of doing backfilling:

Conservative backfilling [42]: The requests are selected in a way such that none of the jobs that were already planned are postponed.

EASY backfilling [36]: It selects a job only if by doing so the next job in the queue is not postponed, even if others are.

In figure C.3 we can see the result of scheduling using FCFS (a) and also the backfilled version of it (b).

Other enhancement is the **time prediction** this is similar to techniques like data mining [41], the runtime estimate given by the user is ignored in order to make the schedule [52] and instead a runtime prediction is made based on the behavior of previous jobs.

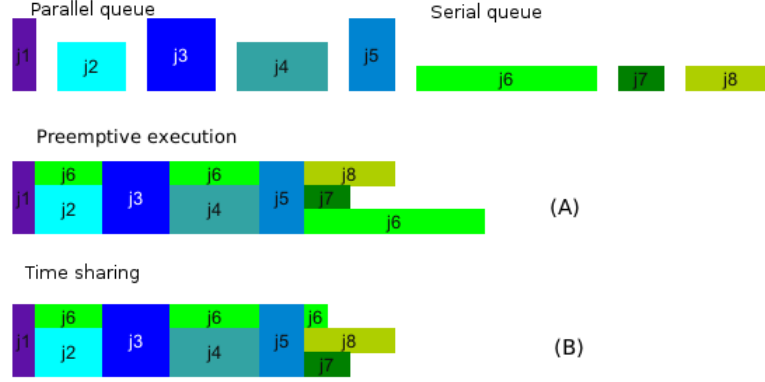


Figure C.4: A) The job runs, if the time is not enough it's killed and requeued
 B) The job is saved to the disk and resumed again.

There are many ways of doing the predictions, one is using the whole history of the jobs, or using only the few last of them, calculating the median, or using the same that lasted the earliest one having finished, etc.

Multitasking is another enhancement that allows many jobs to run in a single processor [19], this kind of solution however, could affect the performance of the jobs [8], how much it does, we don't know, however the solution proposed in this work share some common properties with this kind of enhancement.

The **preemptive execution** C.4a allows a job to run as long as there are enough resources available even if there is not enough time for it to finish running, if the job happens to finish in time then it's good, otherwise it is killed and queued again.

Time sharing C.4b is like an enhancement of the previous one, in this case, if the job doesn't finished running, then it's saved to the disk and resumed again later, this way the work done before is not waisted away.

Appendix D

Graphs

Here a set of graphs with very little relevance for this work that some people may find interesting.

Figure D.1 is the base experiment where no VMs are running at all, that is, the classical cluster; this experiment applies as starting point for the other two; the *X* axis is the number of the node (101 through 104 real), the *Y* axis is the *number of jobs* in the *leftmost* graph and the *time* in the *rightmost* graph.

Figures D.2 through D.7 are extracts from the experiment where 1 VM was created per node every 4 minutes, the last 4 nodes are the physical ones and the first *n* are VMs running in any random node; the upper VM limits are 1, 2, 4, 8, 12 and 16 respectively, in the graph of the left side we see the amount of jobs that each node (physical or virtual) ran, while in the right side we see the average runtime of the jobs that ran in each of those nodes.

Figures D.8 through D.13 are the same as the previous but this time making all the VMs available from the very beginning of the experiment.

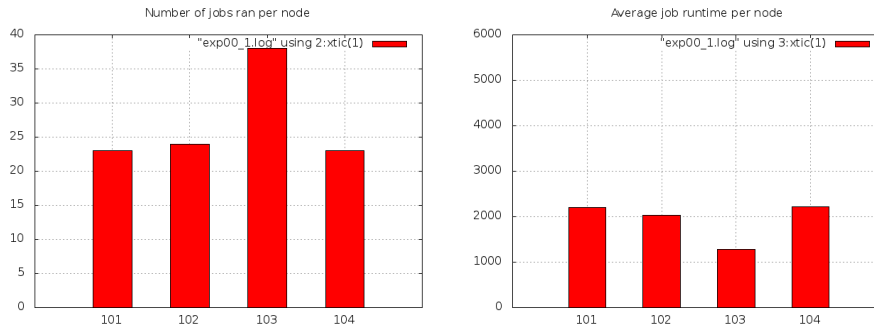


Figure D.1:

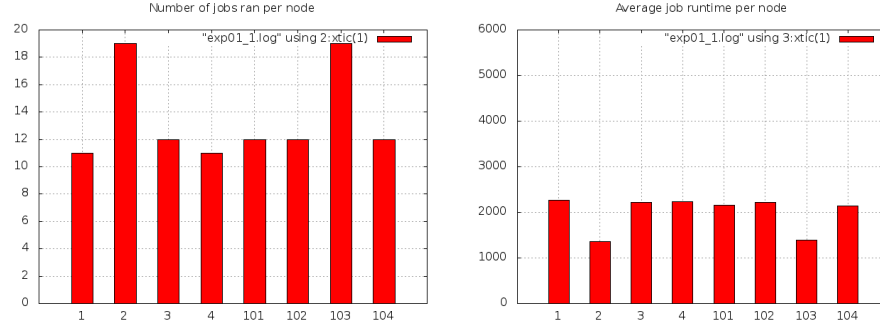


Figure D.2:

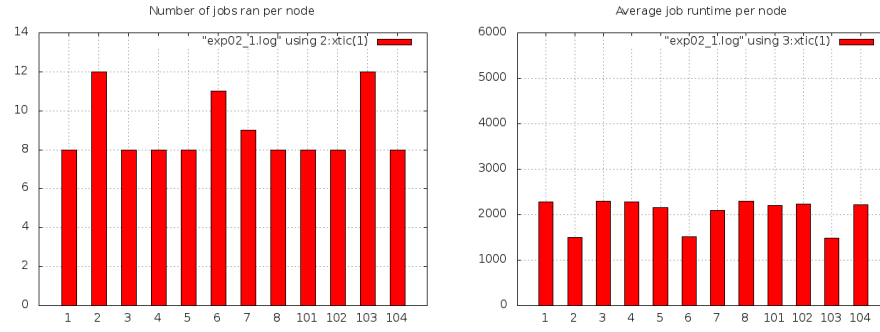


Figure D.3:

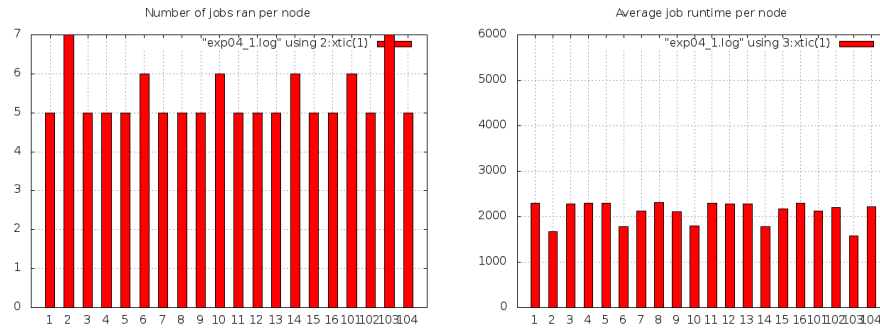


Figure D.4:

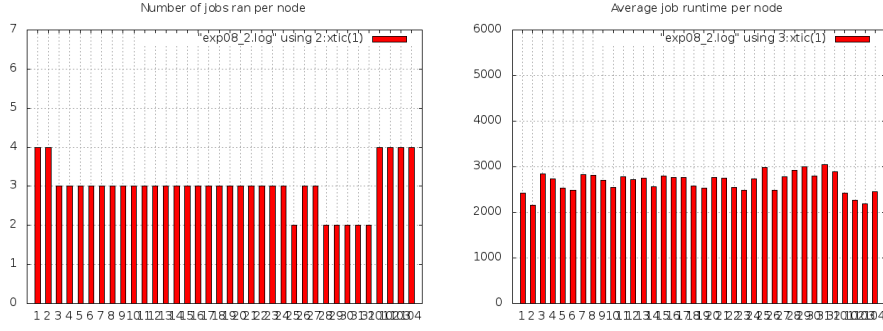


Figure D.5:

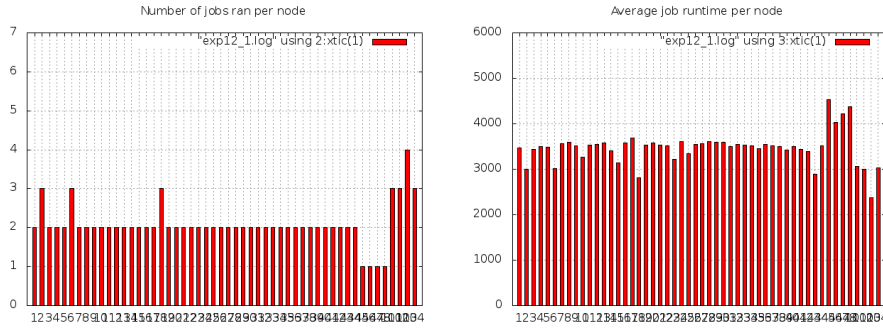


Figure D.6:

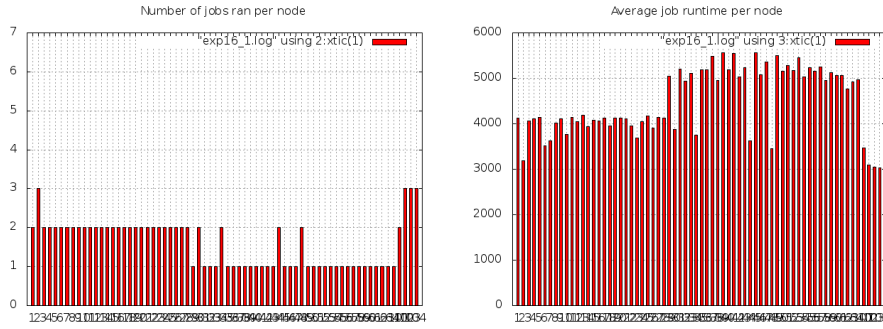


Figure D.7:

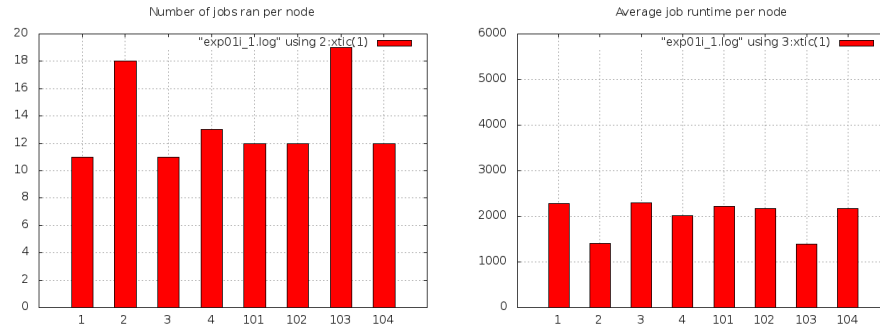


Figure D.8:

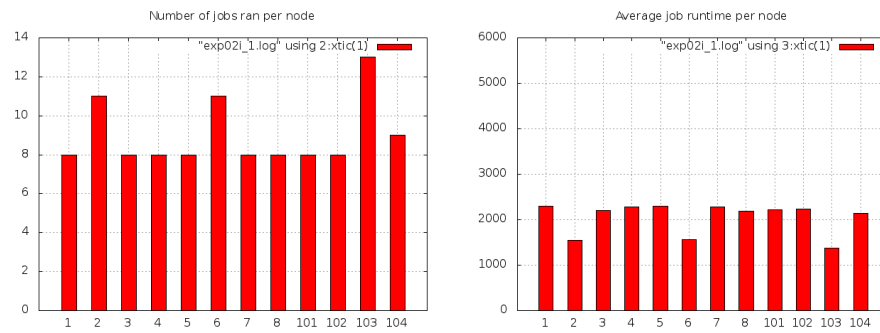


Figure D.9:

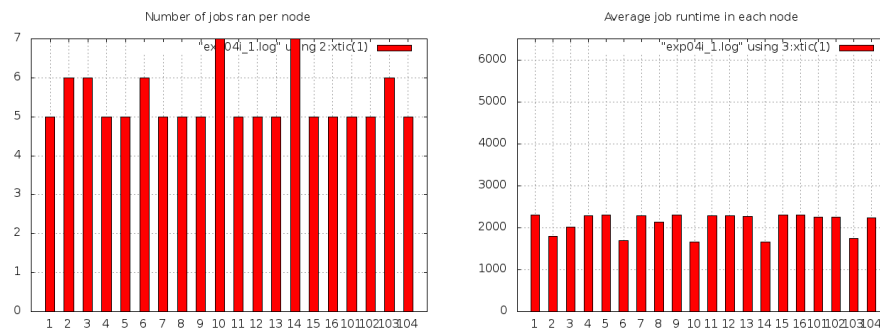


Figure D.10:

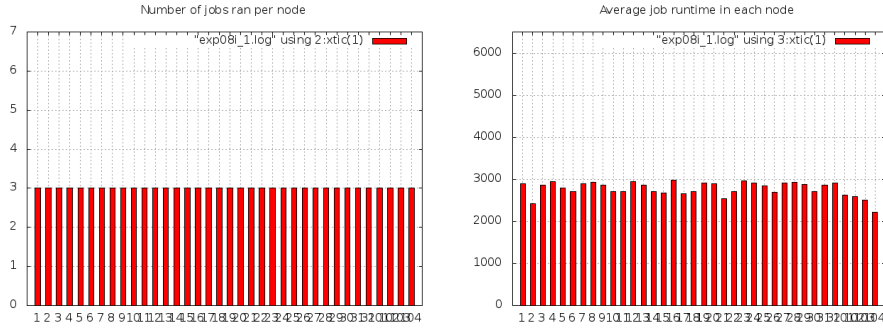


Figure D.11:

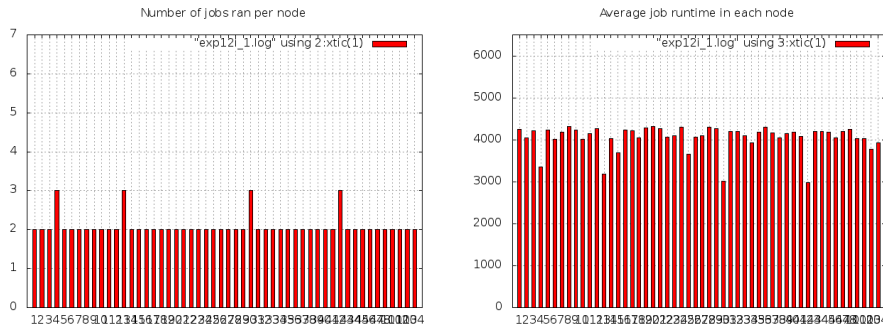


Figure D.12:

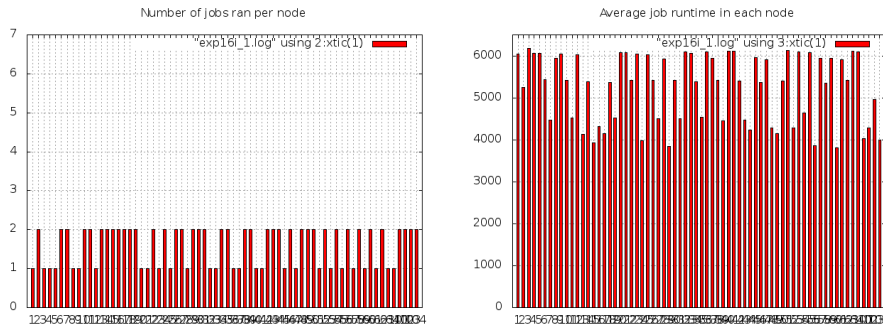


Figure D.13:

Appendix E

Source code

Here are included some pieces of the software developed for this work: Thiao. Only a few functions and files that we consider important are shown.

The complete source code of the project is freely available under the GNU/gpl license at <https://github.com/scarmiglione/thiao>.

rc.local: Sets the appropriate hostname and IP address of the VM, depends on pre-rc.sh and post-rc.sh:

```
#!/bin/bash
mount /mnt
if [ -f /mnt/pre-rc.sh ]
then
    echo cargando el contexto
    . /mnt/pre-rc.sh
else
    echo no hay contexto
    exit -1
fi
echo $HOSTNAME > /etc/hostname
hostname $HOSTNAME
if [ -n "$IPO" ]; then
    ifconfig eth0 $IPO
fi
if [ -n "$GATEWAY" ]; then
    route add default gw $GATEWAY
fi
if [ -f /mnt/post-rc.sh ]; then
    bash /mnt/post-rc.sh
fi
```

pre-rc.sh: reads the environment file `context.sh` sent by OpenNebula and sets some variables needed by it.

```
#!/bin/bash
if [ -f /mnt/context.sh ]; then
    . /mnt/context.sh
fi
if [ -n "$IP" -a -n "$HN_BASE" -a -z "$IP0" ]; then
    IP0=$IP
    IP=${IP:'expr index $IP .' }
    IP=${IP:'expr index $IP .' }
    IP=${IP:'expr index $IP .' }
    HOSTNAME=$HN_BASE$IP
    #echo siiii
fi
```

post-rc.sh: updates some configuration files and start required services.

```
#!/bin/bash
con=/mnt
if [ -f /mnt/context.sh ]
then
    echo cargando el contexto
    . /mnt/context.sh
else
    echo no hay contexto
    exit -1
fi
if [ -f $con/slurm.conf ]; then
    cp $con/slurm.conf /etc/slurm-llnl/
fi
if [ -f $con/munge.key ]; then
    cp $con/munge.key /etc/munge/
fi
if [ -f $con/prolog.sh ]; then
    cp $con/prolog.sh /usr/local/bin/
    chmod a+x /usr/local/bin/prolog.sh
fi
for i in rc.local hosts resolv.conf fstab groups group
        shadow passwd ;
do
    if [ -f $con/$i ]; then
        cp $con/$i /etc
    fi
done
/etc/init.d/slurm-llnl restart
```

remote.cpp

Receives a command with it's parameter list and forward it to the server.

```
bool one_rpc(const string service, xmlrpc_c::paramList *params,
             xmlrpc_c::value *rpc_value){
    xmlrpc_c::clientSimple client;
    xmlrpc_c::value result_rpc;
    xmlrpc_c::paramList param;
    unsigned int i;

    param.add(xmlrpc_c::value_string(rpc_id));

    if (params != NULL)
        for (i=0; i<params->size(); i++)
            param.add((*params)[i]);

    client.call(serverUrl, service, param, &result_rpc);

    if ( result_rpc.type() != result_rpc.TYPE_ARRAY ){
        cout << "expected an array but got something else,
                aborting: " << result_rpc.type() << endl;
        return false;
    }

    xmlrpc_c::value_array result_array(result_rpc);
    vector<xmlrpc_c::value> const result(
        result_array.vectorValueValue());
    xmlrpc_c::value_boolean status = static_cast<xmlrpc_c::value>
        (result[0]);

    *rpc_value = result[1];

    return static_cast<bool>(status);
}
```

Retrieves the list of active VMs and store them in a list for further processing.

```
void fill_vm_list(vector<VirtualMachine*> *vms){
    xmlrpc_c::clientSimple client;
    xmlrpc_c::value result_rpc;
    string const service = "one.vmpool.info";

    client.call(serverUrl, service, "si", &result_rpc,
                rpc_id.c_str(), -2);

    if ( result_rpc.type() != 6 ){
        cout << "expected an array but got something else,
```

```

        aborting: " << result_rpc.type() << endl;
    return;
}

xmlrpc_c::value_array result_array(result_rpc);
vector<xmlrpc_c::value> const result(
    result_array.vectorValueValue());
xmlrpc_c::value_boolean status = static_cast<xmlrpc_c::value>
    (result[0]);
xmlrpc_c::value_string vm_xml =
    static_cast<xmlrpc_c::value_string>(result[1]);

if ( static_cast<bool>(status) ){
    string xml_str = static_cast<string>(vm_xml);
    TiXmlDocument root;
    root.Parse(xml_str.c_str());
    TiXmlNode *element = root.FirstChild("VM_POOL")->
        FirstChild("VM");
    while(element != NULL){
        cout << "VM found" << endl;
        VirtualMachine *vm = new VirtualMachine(element);
        vms->push_back(vm);
        element = element->NextSibling("VM");
    }
} else {
    cout << "RPC call failed to retrieve VM list" << endl;
    cout << static_cast<string>(vm_xml) << endl;
}
}
}

```

resume.cpp launches the VMs specified as parameters

```

int main(int argc, char ** argv){
    sqlite3 *db;
    list <string> hlist, running_vms;
    int rc;
    char *errmsg = NULL, oneid[10];
    bool force_vm_creation = false;

    if (argc < 2 || argc > 3){
        usage();
        return 1;
    }

    if (argc == 3){

```

```

    if ( argv[1] == string("-f") || argv[1] == string(
        "--force" ) ){
        force_vm_creation = true;
        hlist = extend_host_list(argv[2]);
    } else if ( argv[2] == string("-f") || argv[2] == string(
        "--force" ) ) {
        force_vm_creation = true;
        hlist = extend_host_list(argv[1]);
    } else {
        usage();
        return 1;
    }
} else
    hlist = extend_host_list(argv[1]);
putenv((char*)one_auth);

rc = sqlite3_open((char*)db_file, &db);
if (rc){
    cout << "Couldn't open the database, aborting..." << endl;
    sqlite3_close(db);
    return 1;
}

rc = sqlite3_exec(db, "PRAGMA journal_mode = OFF", NULL, 0,
    &errmsg);

while ( ! hlist.empty() ){
    while ( !running_vms.empty() ) running_vms.pop_back();
    rc = sqlite3_exec(db,
        (oneid_from_hostname + hlist.front() + "'")
        .c_str(),
        make_list_callback, (void*)&running_vms,
        &errmsg);

    if ( running_vms.empty() || force_vm_creation ){
        if ( !running_vms.empty() && force_vm_creation )
            cout << "forcing the creation of " <<
                hlist.front() << endl;

        ifstream t((vm_script_dir + "/" + hlist.front() +
            ".one").c_str());
        string vm_def((istreambuf_iterator<char>(t),
istreambuf_iterator<char>()));

        xmlrpc_c::value value;
        xmlrpc_c::paramList param;
        param.add(xmlrpc_c::value_string(vm_def));

```



```

        if ( one_rpc("one.vm.allocate", &param, &value) ){
            xmlrpc_c::value_int vi =
static_cast<xmlrpc_c::value_int>(value);
            sprintf(oneid, "%d", static_cast<int>(vi));
            cout << "oneid: " << oneid << endl;
        }
        else{
            cout << "error trying to create the VM," << endl;
            return 0;
        }

        string query = register_hostname_oneid +
            hlist.front() + string("'", " ) + oneid + string(")");
        cout << query << endl;
        rc = sqlite3_exec(db, query.c_str(), NULL, 0, &errmsg);
    } else {
        cout << "This VM seems to exists: " << hlist.front()
            << " NOT starting it again, "
            << "you can change this with --force." << endl;
    }

    hlist.pop_front();
}
return 0;
}

```

balancer.cpp migrates the VMs between the nodes to make the load balancing.

```

int main(int argc, char **argv){
    vector<class Host*> hlist;
    vector<class VirtualMachine*> vms;
    list<class Host*> sorted;
    class Host *hl;
    unsigned int i = 0, j;

    fill_host_list(&hlist);

    while( i < hlist.size() ){
        hl = hlist[i];
        cout << hl->getName() << endl;
        sorted.push_back( hlist[i] );
        i++;
    }
}

```

```

sorted.sort(compare_host_load);
for (i=0; i<hlist.size(); i++){
    hlist[i] = sorted.front();
    sorted.pop_front();
    cout << "used: " << hlist[i]->getUsedCpu() <<
        " percentage: " <<
        hlist[i]->getCpuUsagePercentage() << endl;
}

fill_vm_list(&vms);
for (i=0; i<vms.size(); i++){
    cout << "vmid " << vms[i]->id << endl;
    cout << "host " << vms[i]->host_id << endl;
}

match_vm_host(&hlist, &vms);

for (i=0; i<hlist.size(); i++)
    if (hlist[i]->getState() != Host::NODE_ON){
        cout << hlist[i]->getName() << " --> is offline,
            dropping from the list" << endl;
        hlist.erase(hlist.begin()+i);
        i--;
    }

i = 0;
j = hlist.size() - 1;
while (i < j){
    hlist[i]->migrateVmFrom(hlist[j]);
    i++; j--;
}
return 0;
}

```