Scheduling and Dynamic Management of Applications over Grids

Ghislain Charrier

▶ **To cite this version:**

Ghislain Charrier. Scheduling and Dynamic Management of Applications over Grids. Networking and Internet Architecture [cs.NI]. Ecole normale supérieure de lyon - ENS LYON, 2010. English. NNT : . tel-00590292

HAL Id: tel-00590292
https://theses.hal.science/tel-00590292

Submitted on 3 May 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

N° d'ordre : 601
N° attribué par la bibliothèque : 10ENSL601

- ÉCOLE NORMALE SUPÉRIEURE DE LYON -
Laboratoire de l'Informatique du Parallélisme

THÈSE

en vue d'obtenir le grade de

Docteur de l'Université de Lyon - École Normale Supérieure de Lyon

Spécialité : Informatique

au titre de l'École Doctorale d'Informatique et de Mathématiques

présentée et soutenue publiquement le 3 décembre 2010 par

M. Ghislain CHARRIER

Scheduling and Dynamic Management of Applications over Grids

Directeur de thèse :	M.	Frédéric	DESPREZ
Co-encadrant de thèse :	M.	Yves	CANIOU
Après avis de :	Mme.	Françoise	BAUDE
	M.	Michel	DAYDE

Devant la commission d'examen formée de :

Mme.	Françoise	BAUDE	Membre/Rapporteur
M.	Yves	CANIOU	Membre
M.	Michel	DAYDE	Membre/Rapporteur
M.	Frédéric	DESPREZ	Membre
M.	Pierre	KUONEN	Président
M.	Olivier	RICHARD	Membre

Acknowledgments

First of all, I would like express my dearest gratitude to the jury members of my defense who accepted to evaluate my work of these three years of Thesis: to Pierre Kuonen, who accepted to preside the jury; to Françoise Baude and Michel Dayde for their reviews of my written document as well as their precious remarks; finally to Olivier Richard, for his relevant remarks about my work.

Of course, there were two more members in the jury, the two persons without whom I could not have prepared a Ph.D.: Yves Caniou and Frédéric Desprez, my colleagues, advisers, tutors, bosses, mentors, idols, Gods, ..., and friends. It was really a pleasure working with both of you.

Thank you for the many hours you spend helping me, giving me advices, showing me the proper work direction. Your experiences and knowledge enabled you to always pinpoint problems or incoherences very quickly in my work. This, from the beginning of the Ph.D. until the defense day.

There are also many people from the LIP I would like to thank, especially the (current and ex) LUG members. However, in order not to forget anyone, I will forget everyone in this thesis chapter (but not in my heart).

For those who know me: Yeah, no (very) bad jokes in here, sorry guys.

Contents

List of Figures

CHAPTER 1

Introduction

Contents

This Thesis aims at evaluating and improving scheduling of jobs on Grid platforms composed of several parallel machines. The solutions we propose take into account the different levels of the platform. First, we aim at improving the global scheduling of the Grid by balancing the load among different clusters. From this point of view, we propose and evaluate different heuristics to dynamically schedule the jobs at the global level. Second, at a local level, we also introduce, compare, and implement in a grid middleware, numerous heuristics designed for a specific climatology application used to predict the climate for the next century.

This chapter presents the motivations of the Thesis and the general context. We introduce briefly different architectures of parallel and distributed computing such as the Grid, and detail problems that arise on such platforms. Hence, we describe the problems we tackle by discussing our objectives and contributions.

1.1 Motivation

Moore's law states that the number of transistors on a chip approximately doubles every two years. It stays true up to now. Indeed, a modern coffee brewer uses a chip with more computing capabilities that the first space module that landed on the moon. The Apollo Guidance Computer used a processor running at an "astonishing" frequency of $2.048\,\mathrm{Mhz}$ [100]. However, a fundamental change occurred a few years ago. Previously, a greater number of transistors was used to perform more computations on a single chip by increasing the clock frequency, lengthening the pipeline or adding new specialized operations. Nowadays, manufacturers tend to multiply the number of computing units on a single chip because physical limitations prevent them from increasing the frequency. Thus, it is not possible anymore to wait for a faster processor to solve a problem, but scientists and engineers have to rethink their methods and software to take advantage of the multiprocessing units.

Using multiple computing elements to compute in parallel is new for general public applications. However, parallel and distributed programming have been in use for many years in the context of solving large scientific problems. Scientists in different areas of expertise used it for decades. Meteorologists, physicists, climatologists, biologists, and even computer scientists need to perform complex simulations with bigger models becoming more and more accurate. The gain brought by the increase of transistors in processors was, and is, still not sufficient. Thus, computers manufacturers had to invent computers able to cope with the demands in processing power. They started by simply adding several processors in a single computer and then interconnecting different computers together to perform computations. As time passed, more and more computers were interconnected in order to create the Grid that interconnects geographically distributed resources, followed by the Cloud that gives the illusion of infinite resources, the last trend in distributed environments being "Sky Computing", the interconnection of Clouds. The need of always having more and more powerful computers and faster software regrouped researchers to form the field of *High Performance Computing* (HPC).

In parallel platforms used in HPC such as clusters or parallel machines, computing resources are shared between multiple users that may be part of different institutions. It is thus very important to have smart mechanisms to accommodate all the jobs submitted by users and to share the available resources. Resources sharing is done with the help of queuing systems. Their role is twofold. The first role, referred to as *mapping*, is to manage resources, thus providing the jobs with an exclusive access to resources. The other role is to decide when to allocate the resources to a job. This second role is known as *scheduling*. It is usually done using *space sharing* techniques. Each resource is attributed exclusively to a job for a given period of time. Good management policies can provide great improvements on different metrics: maximum utilization of the platform, faster execution times, and better user's satisfaction are objectives that are often considered by the HPC community.

Parallel machines used in HPC may not be sufficient to cope with really large applications. Hence, it is necessary to use different computing resources connected together through high performance network. The aggregation of resources is known as distributed computing. Distributed computing includes platforms such as Grids and Clouds. Grids

interconnect geographically distributed and heterogeneous resources together to take advantage of their overall processing capacity. Managing these distributed resources adds different software and hardware abstraction layers to the Grid. The different layers go from the simple computer to clusters or parallel machines and extend to the Grid which interconnects the different resources via high bandwidth networks. Each layer has its own management system and constraints, thus complicating the usage, monitoring and maintenance of the overall platform. Furthermore, each layer decreases the efficiency of the application. A computer is managed by an operating system, a parallel machine may be managed by a queuing system, and a Grid may be managed by a grid middleware.

A grid middleware is in charge of the execution of jobs that may come from different users. It maps jobs on parallel resources, where the jobs are then managed by a queuing system. The mapping and scheduling may be simple, using a simple round robin selection on all the distributed resources, or can be very complex using advanced scheduling techniques. The grid middleware also monitors the jobs and the resources in order to react properly in case of failure, or to take advantage of a change in the execution environment: new resources may become available or some may disappear. The grid middleware is also in charge of the data of an application. Indeed, the presence of certain data on a cluster may be mandatory for the execution of an application, thus the grid middleware has to make sure that data is available when needed. In the context of a Grid used by several users from different institutions, another role of a middleware may be to authenticate users to ensure that only the authorized persons can access private data.

The grid middleware is supervising jobs that can require some performance guaranties. Thus, it should take into account similar goals as the ones the HPC community defined for the usage of parallel resources. These goals include efficient mapping and scheduling strategies in order to maximize the platform usage or minimize the jobs' execution time in order to improve user's satisfaction. Thus, the grid middleware has to take into account different information coming from both the users and the platform monitoring, in order to provide good scheduling and management strategies. By aggregating all the different sources of information, the grid middleware must be able to manage jobs as efficiently as possible.

In the remaining of this chapter, we present some of the problematics arising when executing applications on distributed platforms shared by different users. With these problematics in mind, we describe the objectives that we aim to achieve in this Thesis, and our contributions. Then, we present the resulting publications of our contributions. We finish by an outline of the remainder of the Thesis document.

1.2 Problematics Related to Scheduling in Distributed Environments

As we introduced before, a grid middleware has to interact with queuing systems [82] deployed on each of the parallel resources such as supercomputers, or clusters of interconnected machines. In order to provide efficient scheduling decisions, a grid middleware should therefore be aware of the queuing system managing the parallel resources. Most

queuing systems give an exclusive access to a job for some time, and postpone the other jobs for the time being. Queuing systems allow multiple jobs to be executed concurrently on different resources if they require less than the total amount of available resources. Several metrics can be studied and used as points of comparison to determine the efficiency of the system. However they are often contradictory: improving one may cause another to worsen. Multi-criteria algorithms exist to take into account multiple metrics at the same time [75] and find a trade-of between possibly opposite objectives. Each queuing system gives priority to the optimization of a specific metric [80]. Knowing which metric is used, the middleware may be able to provide better results.

A grid middleware can base its scheduling decisions on several parameters. The amount of information provided by the users is thus a crucial point. With more information on the jobs to execute, scheduling decisions are expected to be better. However, providing all the expected data is hard, and in some cases impossible. For example, giving an estimation of the execution time of an application may not be possible when the final result does not depend on the size of input data but on the content of this data. Information used to perform the schedule can be an estimation of the execution time of the application with a number of processors used to run the application, in addition to the estimation of internal communications made by the application. The information can also be determined automatically [52, 166]. If the user uses a grid middleware to connect to a remote cluster, the middleware may be able to fill up automatically all or parts of the information needed. Techniques such as data mining are used for this purpose.

Because scheduling is based on information given by the user or determined automatically by the middleware, it should be considered inherently wrong [119]. Indeed, knowing or estimating precisely the execution time of an application in advance is usually not possible. It depends on many parameters, such as the number of other applications running on the resource on which it is executed, the available bandwidth, the type of processor used for the computations, the content of the input data, *etc*. Thus, scheduling decisions are based on wrong values [129]. The way to compensate errors and change decisions that have been made is very important. An efficient middleware in a dynamic system should be able to provide good results, even when good scheduling decisions taken earlier are now proved to be bad scheduling decisions.

Usually, taking into account different clusters or parallel machines is hard because of the heterogeneity between the resources. The number of processors on each cluster can be different, the power and type of processors can change, the amount of memory on each computing element can also be different, and the operating systems may not be the same. Scheduling decisions made by the grid middleware should take into account all these parameters [37, 90]. Dealing with software and hardware heterogeneity automatically is also a challenging problem. If an application requires a particular type of resource, such as a particular operating system, a simple mapping is often enough. However, trying to minimize the execution time of applications on heterogeneous platforms is known to be a very hard problem [32, 69].

Resources inside a cluster or a parallel machine are usually homogeneous and interconnected through a very high speed network. However, an application can be so large that it may need several parallel machines to run. Thus, it is necessary to obtain resources that can

be heterogeneous from different queuing systems at the same time. This problem, known as co-allocation [128], is challenging because it is necessary to have all the resources available at the same time on different clusters. Furthermore, connecting geographically distant clusters often implies the usage of slower networks. The execution time of the application can thus be greatly impacted. Dynamic solutions exist using co-allocation when necessary, but regrouping the execution on a single cluster if enough resources become available [131]. The middleware should be able to monitor the environment and change its decisions dynamically and accordingly to the new state of the platform.

Another common scheduling problem deals with DAG scheduling. Applications may be represented as Directed Acyclic Graphs (DAGs). It represents the decomposition of the application in several tasks with dependency relations. A specific task can not start while all its dependencies have not been executed. This problem is also very well studied in the HPC community. Scheduling such applications is already a difficult problem, and the scheduling of several concurrent DAGs is also very common [61, 105, 134, 175], and even harder.

Some of the previous issues are assessed with generic solutions implemented in middleware [19, 150] able to adapt to different situations and able to deal with different queuing systems [27, 94] on clusters. However, the applications can also take part in the scheduling process. The application may be able to take advantage of some specific situations using internal knowledge to obtain a better scheduling [96]. For example, if the application knows the architecture of the platform, it can optimize its internal communications to have tightly-coupled part of the code close together. Another possibility is for the middleware to take advantage of the knowledge of a particular application. It may thus be able to provide a better scheduling. However, this solution is not generic because it serves only one application, or a class of applications. It is also possible for the application and the middleware to collaborate [38] by exchanging information. Communications between the two entities should lead to better results. If the application has different needs of resources over time, it can send an update of its need to the middleware that can find new resources to assess the needs of the application.

At the uppermost level, one may want to access different Grids, managed by different administrations, transparently. Accessing different Grids usually means accessing different middleware. Middleware have been developed independently and may thus not be compatible with other. Two research directions emerge from this need of compatible middleware systems. A short-term solution is the use of common interfaces, adapters, or gateways [169]. However, these solutions are only working between specific middleware because of the lack of standards. On a longer term, groups such as the Open Grid Forum [1] work on the development of new standards that can be used by different middleware thus providing interoperability [148].

1.3 Objectives and Contributions of this Thesis

The context of this Thesis is a Grid composed of several clusters, where clusters are interconnected through a high bandwidth network (WAN for example). Such architectures

1. http://www.ogf.org

are very common in the Grid domain. As we previously stated, the management of the execution of jobs in such a distributed environment relies on the ability of the grid middleware to choose the appropriate resources, and to dynamically adapt to changes on the platform by modifying previous scheduling decisions. Because the multi-cluster architecture is common, we want to study the problem while keeping untouched the underlying parallel machines or clusters as well as their resource managers. We want to provide a software layer to manage jobs as efficiently as possible on top of the existing environment. By improving the jobs management, we hope to improve the overall efficiency of the platform. A better efficiency will enable the platform to execute more jobs for a given duration. If more jobs are executed, all the users should benefit from the gain, thus improving their satisfaction.

In order to manage resources efficiently, we propose different scheduling policies. We mainly focus on two scheduling problems in this context: the automatic reallocation of jobs in a Grid and the design of scheduling heuristic for a specific application.

The first focus of our work is to propose a generic solution to dynamically adapt to errors on runtime estimates given to resource management systems. As pointed out in the previous section, determining the runtime of a job can be very hard. Scheduling algorithms used by resource managers on clusters generally base their decisions on the estimation of the execution time. Thus, errors made on these estimations have a direct impact on the scheduling of the cluster. When a job finishes earlier than expected, the jobs after it in the waiting queue may be started earlier. This local change impacts the global scheduling decision made by the grid middleware. If new resources become available, jobs already in the system should benefit from them. In order to take advantage of this, we propose a reallocation mechanism to move jobs between clusters. In the architecture we propose, we use the grid middleware as an intermediary layer between clusters. It takes scheduling decisions regarding the assignment of jobs to cluster and automatically manages them once submitted to the local resources management systems of the clusters. By moving jobs between clusters, we hope to improve the global efficiency of the Grid.

We study the problem of reallocation in a multi-cluster environment with different kinds of jobs. First, we look at the reallocation of *rigid jobs*: the number of processors to execute a job is defined by the user prior to its execution. Many applications are configured to be executed on a predetermined number of processors. Secondly, we extend our study to *moldable jobs*: a moldable job can be executed on different number of processors, but once the execution starts, this number cannot be changed anymore. Typical parallel applications can take the number of processors as input parameters.

When a user submits a job to the grid middleware, the estimation of the runtime is determined automatically and the mapping of the job on a cluster is done dynamically. A job submitted through the middleware may be moved across the platform as long as its execution has not started. Concerning this reallocation mechanism, we study two algorithms. One algorithm tries to reallocate jobs by comparing their estimated completion time. When a job is expected to finish earlier on another cluster, it is moved there. The other algorithm cancels all waiting jobs and make the mapping again. Both algorithms are tested with different selection policies that choose in which order jobs should be reallocated. We use real-life traces of clusters and show the improvements that reallocation can bring on differ-

ent metrics including the average response time. The response time is the duration that a job spent in the platform. Results on this part led to four publications in international and national conferences [C3, C4, C5, N1].

In our reallocation scheme, the scheduling of jobs is done at two levels. At the Grid level, jobs are mapped on parallel resources by the grid middleware. Then, jobs are managed by the resource managers to be scheduled on the resources of the parallel machine. In order to improve the scheduling of jobs in a slightly different context, we study the scheduling policies of a specific moldable application namely Ocean-Atmosphere. Our goal is still to maximize the efficiency of the platform, but in this case by minimizing the execution time of the application. To schedule it on heterogeneous clusters, the grid middleware is in charge of mapping the application on different clusters, and on each cluster, we develop specific scheduling heuristics to divide the available resources. The middleware solution can be deployed on top of existing clusters, without modifying the underlying architecture.

The second focus of our work is therefore to propose efficient scheduling policies to execute a specific application on a multi-cluster Grid. The application, called Ocean-Atmosphere, is an iterative parallel program used to simulate the climate evolution for the centuries to come. It is launched several times in parallel in order to study the impact of parameters on the results. Our goal is to minimize the total execution time of the application with different parameters by taking advantage of the parallel and distributed architecture. Our objective is also to implement the best heuristics in a grid middleware so that climatologists can perform real experiments on the Grid. With the use of our implementation maximizing the performance, climatologists can have more results to study.

Thus we need to design efficient scheduling policies. Therefore, we start by defining a model of the application to represent it as a DAG. Then we formulate different heuristics so as to execute the application on a single cluster. The heuristics divide the processors into groups, and each group executes an instance of Ocean-Atmosphere code. The size of the groups of processors is chosen to obtain the minimum overall execution time. With these heuristics, several instances of the application can be executed in parallel on a cluster, and using the model enables us to obtain an estimation of the duration needed to execute these instances of the application which can be used to submit for batch submissions.

With the estimations of the execution of several instances of Ocean-Atmosphere on a cluster, the middleware can split the instances of the application to execute them on several clusters. The split is done using the estimations and it minimizes the total execution time. The approach is a greedy one, so that the scheduling phase is fast.

The different scheduling policies at both levels are compared using simulations. Heuristics obtaining the minimum execution time have been selected and implemented within the DIET GridRPC middleware. In order to validate our model, we compare the estimations and the real execution times obtained with real-life experiments running on GRID'5000, a research Grid deployed in France over 9 sites. This implementation is also freely available and can be used by climatologists using Ocean-Atmosphere to run large experiments in Grids. Results on this topic lead to two publications in international conferences [C1, C2] and a poster presentation in an international workshop [P1].

1.4 Publications

Our work has been published in several international and national conferences with reviewing committees. More detailed versions of the submitted work were published as INRIA and LIP research reports [2].

Articles in international refereed conferences

[C5] Yves Caniou, Ghislain Charrier and Frédéric Desprez. Evaluation of Reallocation Heuristics for Moldable Tasks in Computational Grids. In the *9th Australasian Symposium on Parallel and Distributed Computing (AusPDC 2011)*. January 17-20 2011, Perth. Australia. Note: To appear.

[C4] Yves Caniou, Ghislain Charrier and Frédéric Desprez. Analysis of Tasks Reallocation in a Dedicated Grid Environment. In the *IEEE International Conference on Cluster Computing 2010 (Cluster 2010)*, pages 284–291. September 20-24 2010, Heraklion, Crete. Greece.

[C3] Yves Caniou, Eddy Caron, Ghislain Charrier and Frédéric Desprez. Meta-Scheduling and Task Reallocation in a Grid Environment. In the *Third International Conference on Advanced Engineering Computing and Applications in Sciences (ADVCOMP'09)* (acceptance rate: 28%), pages 181–186. October 11-16 2009, Sliema. Malta.

[C2] Yves Caniou, Eddy Caron, Ghislain Charrier, Frédéric Desprez, Éric Maisonnave and Vincent Pichon. Ocean-Atmosphere Application Scheduling within DIET. In the IEEE *APDCT-08 Symposium. International Symposium on Advanced in Parallel and Distributed Computing Techniques*, pages 675–680. Held in conjunction with ISPA'2008. December 10-12 2008, Sydney. Australia. Note: Invited paper from the reviewed process of ISPA'08.

[C1] Yves Caniou, Eddy Caron, Ghislain Charrier, Andréea Chis, Frédéric Desprez and Éric Maisonnave. Ocean-Atmosphere Modelization over the Grid. In the IEEE *37th International Conference on Parallel Processing (ICPP 2008)* (acceptance rate: 30%), pages 206–213. September 9-11 2008, Portland, Oregon. USA.

Articles in national refereed conferences

[N1] Ghislain Charrier and Yves Caniou. Ordonnancement et réallocation de tâches sur une grille de calcul. In the *19èmes Rencontres francophones du Parallélisme (RenPar'19)* (acceptance rate: 64%). September 9-11 2009, Toulouse. France.

Posters in reviewed international conferences

[P1] Ghislain Charrier. From Scheduling Theory to Practice: a Case Study. In the *ACM/IEEE Workshop of Doctoral Students of the 5th International Conference on Soft Computing as Transdisciplinary Science and Technology (CSTST-SW'08)*, pages 581–582. October 2008, Cergy-Pontoise. France.

Research reports

2. We only present the references for INRIA reports, as LIP reports are identical

[R4] Yves Caniou, Ghislain Charrier and Frédéric Desprez. Evaluation of Reallocation Heuristics for Moldable Tasks in Computational Dedicated and non Dedicated Grids. Research report RR-7365, *Institut National de Recherche en Informatique et en Automatique (INRIA)*. August 2010.

[R3] Yves Caniou, Ghislain Charrier and Frédéric Desprez. Analysis of Tasks Reallocation in a Dedicated Grid Environment. Research report RR-7226, *Institut National de Recherche en Informatique et en Automatique (INRIA)*. March 2010.

[R2] Yves Caniou, Eddy Caron, Ghislain Charrier, Frédéric Desprez, Éric Maisonnave and Vincent Pichon. Ocean-Atmosphere Application Scheduling within DIET. Research report RR-6836, *Institut National de Recherche en Informatique et en Automatique (INRIA)*. February 2009.

[R1] Yves Caniou, Eddy Caron, Ghislain Charrier, Andréea Chis, Frédéric Desprez and Éric Maisonnave. Ocean-Atmosphere Modelization over the Grid. Research report RR-6695, *Institut National de Recherche en Informatique et en Automatique (INRIA)*. October 2008. Note: also available as LIP Research Report RR2008-26.

1.5 Organization of the Thesis

The rest of this Thesis is organized as follows:

- Chapter 2 presents the necessary background to the correct understanding of this document. First, it describes the evolution of the High Performance Computing systems. Then, the chapter introduces scheduling solutions that can be applied at different levels in a HPC platform. Finally, the chapter describes different tools and platforms used by researchers and scientists to execute experiments in Grids.
- Chapter 3 presents the approach we developed for the task reallocation in heterogeneous Grid environments. We study task reallocation across multiple homogeneous and heterogeneous clusters. We realize this study on two types of parallel tasks: rigid applications as well as moldable ones. We propose a solution at the middleware level to automate the execution while improving the average execution time. We compare different reallocations heuristics with different policies using simulations on real-life workload traces.
- Chapter 4 presents the study of the execution of Ocean-Atmosphere over the Grid. We propose different heuristics to schedule a climatology application on a cluster. We then extend this work by presenting an algorithm to execute the application on an heterogeneous Grid by reusing the heuristics developed for the clusters. We evaluate the heuristics by simulation, and we propose a real implementation of the best heuristics in a middleware. We also compare the simulations with real-life experiments.
- Chapter 5 concludes the Thesis. It summarizes the contributions, gives some final thoughts on the work, and possible future works.

CHAPTER 2

Background

Contents

This chapter aims at presenting the necessary elements to fully understand the achievements of the Thesis for both results around task reallocation and the ones on scheduling for Ocean-Atmosphere. These two studies have the same experimental contexts and the solutions designed for both problems have similar architectures.

This chapter starts by a brief overview of the evolution of the computing resources made available to scientists. First, resources were just simple multiprocessor machines, and they evolved to clusters and parallel machines, then to Grids and they continued towards Cloud and Sky Computing platforms. Then, we introduce scheduling and we present scheduling problems that arise at different levels in a Grid environment. Finally, we present Grid platforms and different tools used in this Thesis: we worked with batch schedulers, middleware, and simulation toolkits.

2.1 Parallel and Distributed Computing Architectures

Understanding the evolution of parallel and distributed architectures helps to understand better the problems related to scheduling on parallel machines and distributed environments. To execute parallel applications, manufacturers had to provide adequate hardware to scientists. The solution they provided is to put an increasing number of computing elements in computers. A computer embedding more than one computing element is said to be a parallel machine. When it is not possible to increase the number of computing elements in a single machine, several distributed parallel machines are interconnected. This hierarchical structure reflects the way scheduling is done in parallel and distributed environments.

In this section, we give a brief tour of existing architectures, going from the simple multiprocessors computers, to supercomputers, leading to distributed environments such as Grids and Clouds. Each kind of architectures extends another to reach a larger computation capability. Therefore, we present the hierarchical structure of the different platforms.

2.1.1 Multiprocessor Computers

The first category of parallel machines are Symmetric Multiprocessing (SMP) computers. A computer enters this category if all the computing elements share the same data using a single main memory and if all the computing units are managed by a single operating system. Figure 2.1 represents the architecture of a SMP computer. Most modern operating systems are able to take advantage of this architecture [151].

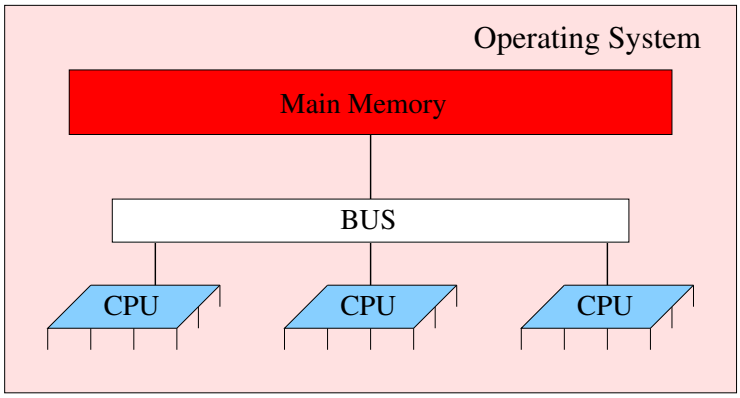


Figure 2.1 – The SMP architecture.

SMP architectures are well adapted to run small scale parallel applications. Each thread or process can run on a computing unit while all processes have access to the same data through the main memory. The application benefits of the parallelism as well as the avoidance of context switching inherent to the execution of several application in monoprocessor architectures. However, programming on an SMP architecture requires other paradigms [97] than just simple sequential development and thus makes the programming more difficult. Higher level APIs such as OpenMP [68] are aiming at reducing the difficulty of programming of such platforms.

Having a single memory space enables easy data sharing among the computing elements. However the processing units are able to process data faster than the memory can store and distribute it. With the increase in processing speed, the memory accesses were not able to keep up with the demands of the computing elements. Thus, on such platforms, the number of processors that can be used is limited. To cope with this bandwidth limitation, new architectures were developed.

To overcome these limitations, Non-Uniform Memory Access (NUMA) computers were developed. The bandwidth required by the memory and the buses to transfer the data fast enough to the processors is not attainable in practice, thus NUMA introduces distributed memory access [26]. Several memory banks are connected to different processors in a hierarchical manner, but every processor can access all memory through a fast communication link. Figure 2.2 schematizes this distributed memory.

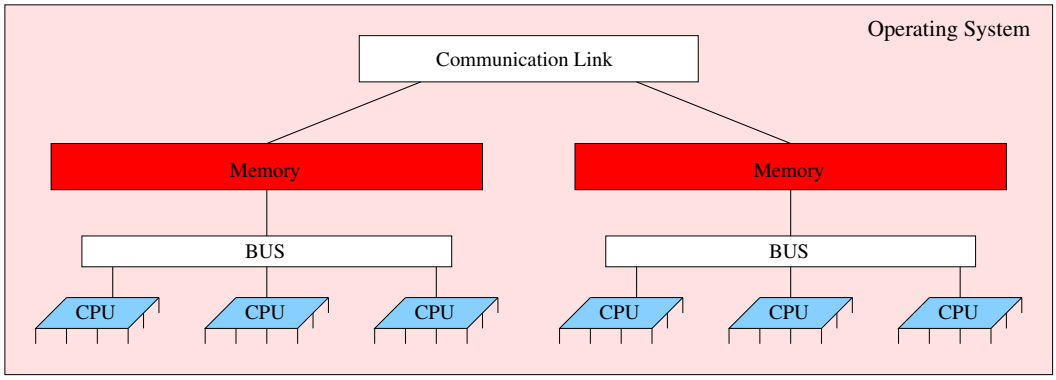


Figure 2.2 – The NUMA architecture.

Each process has access to all memory. However, depending on the data location, the time needed to gather the data locally may vary a lot. Important latencies may arise if the data is located on another memory bank.

Programming NUMA architectures may be harder than SMP. To obtain maximum performance, the user may want to manage the data location in memory himself [133] which increases programming difficulty. However, the main operating systems (Linux [1], Windows [2], Solaris [3], ...) support NUMA architectures, so it is possible to program applications as if it was on a SMP computer but without taking advantage of data localization in memory.

2.1.2 Clusters

A cluster of computers represents a group of different computers tightly coupled to work together [39]. Each computer in a cluster is known as a node and is managed by a

1. http://oss.sgi.com/projects/numa/
2. http://developer.amd.com/pages/1162007106.aspx
3. http://hub.opensolaris.org/bin/view/Community+Group+performance/numa

single operating system that may not be aware of its belonging to the cluster. A node in a cluster is usually a SMP machine. Figure 2.3 presents a usual view of a cluster.

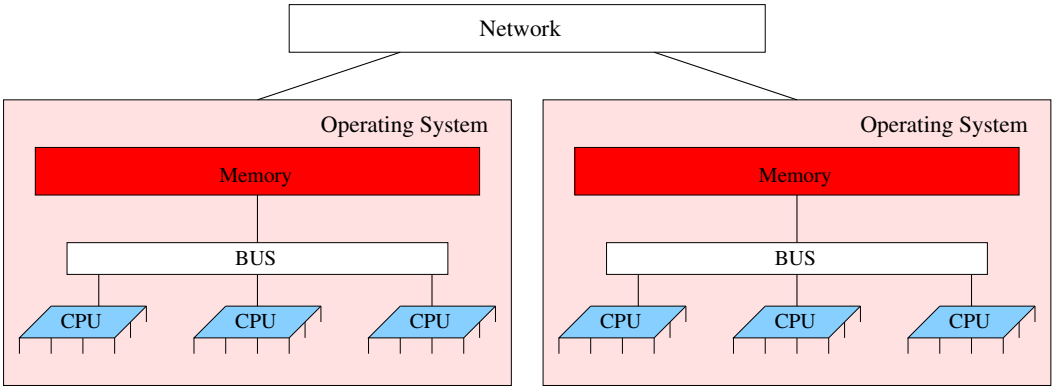


Figure 2.3 – Architecture of a cluster.

Clusters do not possess a central memory easily accessible by all the processes. However, some clusters use distributed file systems such as NFS, but this kind of solution may be very slow. From the application point of view, the memory accesses between the different nodes are left to the programmer of the parallel application. For example, the most common way used by programmers to transmit data between nodes is the Message Passing Interface (MPI) [98]. This standard API provides different communication methods to send or ask for data across different processes distributed among the nodes. Another well known communication library is the Parallel Virtual Machine (PVM) [93].

A cluster is a grouping of independent computers. Thus it is cheaper (initial cost and maintenance), more extensible, and more reliable than a multiprocessor server. Indeed, if a node fails, the rest of the cluster continues to work properly. Furthermore, a cluster is far more extensible than a single computer. Nodes are not aware of each other, so it is theoretically possible to add as many as possible. Some clusters can reach several thousand computing nodes.

2.1.3 Supercomputers

Supercomputers are computers at the cutting edge of technology – at the time of their construction – designed to perform large computations. Early supercomputers from the 80's embedded between four and sixteen processors. This number keeps growing since then and can now reach a few hundred of thousand cores. A general ranking of super computers is kept in the Top500 [17]. This ranking is done according to the floating point operations per second (FLOPS) achieved by supercomputers. In June 2010, the fastest computer is the Cray Jaguar with a peak performance at 1.759 PFLOPS attained with 224,256 cores.

To reach such a high number of processors, modern supercomputers are mostly tightly coupled and highly tuned clusters. Communications have a very important role in such machines, so complex network topologies are deployed to reach the best performance. For

example, the Blue Gene/L supercomputer uses a three-dimensional toroidal network for communications [18].

Supercomputers operating systems in the Top500 are dominated by Unix-like systems as shown in Figure 2.4 taken from WIKIPEDIA [4]. We can see that Unix or GNU/Linux have kept more than 90% of the share since 1994.

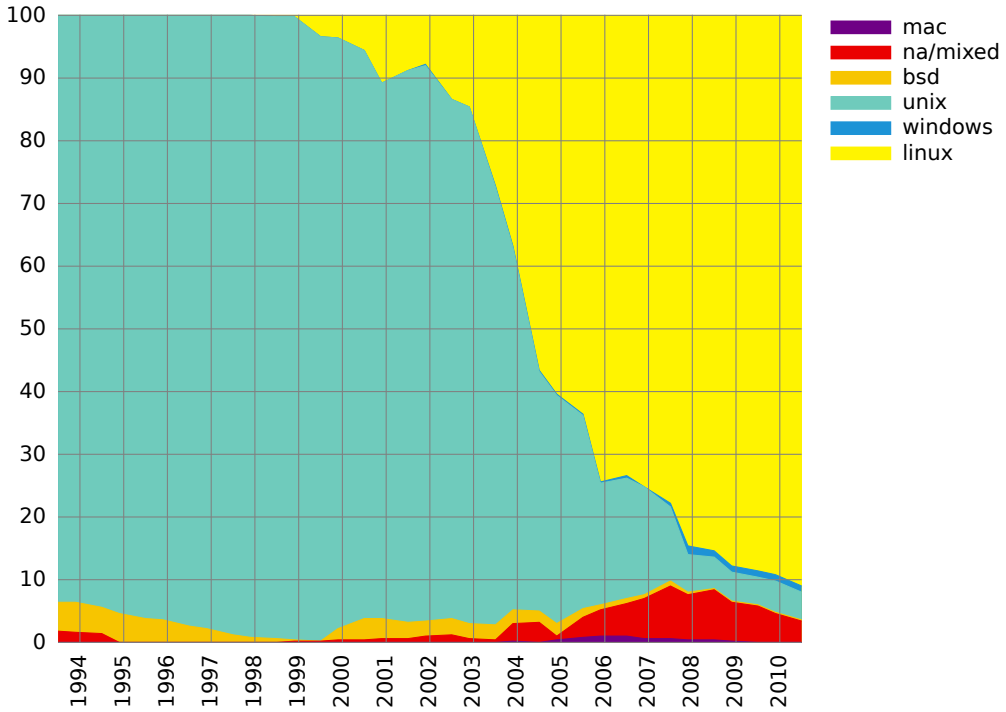


Figure 2.4 – Operating systems used in Top500.

Supercomputers may be specialized to excel in one domain. It is done by providing specific hardware chips to solve a given problem. The most famous example of a specialized supercomputer is *Deep Blue* [42]. It was manufactured by IBM and was specially designed to play chess. It was able to beat Garry Kasparov in a game in 1997.

While supercomputers are well adapted to solve large problems, their cost is very prohibitive. The hardware itself is very expensive, but the maintenance is even more. It is not unusual to build an entire building to hold a supercomputer. This building should be able to sustain the power needs of the machine as well as providing the appropriate cooling infrastructure required to avoid overheating. Because of this, only large institutes and companies can afford supercomputers.

4. Source: http://en.wikipedia.org/wiki/File:Operating_systems_used_on_top_500_supercomputers.svg

2.1.4 Grid Computing

In order to enable institutes with limited financial resources to use large computing power, the aggregation of distributed and heterogeneous resources is necessary. Foster and Kesselman defined the term of *Computational Grid* in [87] to represent this distributed architecture. This name comes from an analogy with the electrical power Grid where anyone has access to electricity without knowing its provenance. In Computational Grids, the electricity is replaced by the computing resources available such as processing power or storage.

In its first definition given by Foster and Kesselman, the Grid was described as *"a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities."*. This infrastructure is owned by multiple organizations and there is not a single entity controlling the entire Grid.

While the Grid was defined as a aggregation of resources owned by different organizations, the first definition was unclear about resource sharing policies among organizations. Thus, the definition was extended by the authors in [88] to address this issue by introducing the concept of Virtual Organization (VO): *"Grid computing is concerned with coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations."*. A VO is the grouping of several individuals and/or institutes sharing the same rules and resources. All members of a VO should be able to directly access resources transparently even when the physical computer is owned by another member in a distant location.

In [85], Foster summarizes the three main points an architecture must have to be considered as a Grid. Thus, a Computational Grid is a system that:

1. "coordinates resources that are not subject to centralized control ...
2. ... using standard, open, general-purpose protocols and interfaces ...
3. ... to deliver nontrivial qualities of service."

In the HPC community, Grids are often multi-cluster Grids [104, 123]. The multi-cluster setup is the interconnection of parallel machines or clusters through high speed wide area networks as presented in Figure 2.5. In the case of multiple institutions belonging to a single VO, a common practice is for each institute to provide one or more clusters. Each cluster may be managed with different software and scheduling policies. However, the access to any resource is done in a transparent manner with the help of a middleware or with the use of standard APIs. Thus, in this context, the multi-cluster Grid already responds to the three points enumerated above.

The multi-cluster approach is very common in the HPC world, however, there are also other kinds of Grids. The Desktop Grid [116] is also commonly used. The concept behind this kind of Grid is to use the processing power or storage of simple desktop computers all around the world. The power of Desktop Grids come from the number of available computers [160]. Desktop Grids are not fit to run HPC applications because of high latencies and low bandwidth between hosts. However, they are well fitted to run parameter sweep applications with little input/output.

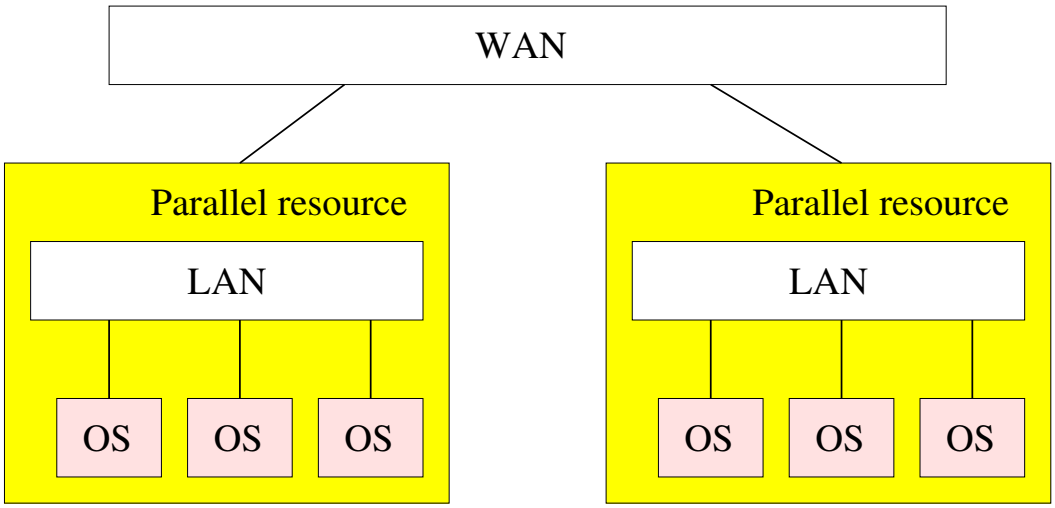


Figure 2.5 – The multi-cluster Grid architecture.

Even if the cost of Grids is lower than supercomputers, it still does not enable individuals or small industries to obtain large resources for a reasonable price. Indeed, to be part of a VO, it is usually mandatory to make available additional computing resources to this VO. Maintaining available resources is still expensive and requires time and qualified people. Deploying and executing an application on a Grid is also quite complicated without being part of a VO.

2.1.5 From Cloud Computing towards Sky Computing

Cloud Computing shares part of the vision of Grid Computing [21]. It aims at reducing the costs of computing and thus enables more users to perform computations. While they share a common vision, Clouds and Grids are different in other aspects [89]. Contrary to Grids, Clouds resources can be accessed on-demand for any amount of time with the illusion of an infinite amount of resources [22]. Furthermore, unlike most Grids, some Cloud architectures let users deploy their own operating systems using Virtual Machines to have a total control on their environment.

Clouds offer different level of services. At the uppermost level is the Software as a Service (SaaS) paradigm. Services are on-demand applications that can be accessed through the Internet. A typical SaaS example is Animoto [2]. It provides an easy way to make animated montages of pictures just by uploading them. Clouds also offer Platform as a Service (Paas) to deliver programming and execution environments. Popular PaaS are the Google's App Engine [4] and Microsoft Azure [6]. Finally, at the lowest level, Clouds may offer the Infrastructure as a Service (IaaS) where users have access to the execution environment through Virtual Machines. Amazon [1] and Nimbus [110] are a well-known IaaS solution. Cloud offer many services, therefore they are sometimes referred to as XaaS: "everything as a service" [120].

In the beginning of Clouds, they were mostly used for simple sequential applications, but recent evolutions enables the HPC community to run parallel applications in the Cloud [77]. Cloud providers such as Amazon with the EC2 [1] (Amazon Elastic Compute Cloud) platform propose new services and new infrastructures. It is now possible to rent high-speed clusters with the "Cluster Compute" [5] instance of EC2 and run HPC applications with good performance comparable to the ones obtained with supercomputers.

The first Cloud providers were renting the computing power of their main servers while they were idle. Indeed, in order to support bursts, companies buy the necessary resources, but they stay idle most of the time. Each company had it's own internal way of managing resources, and their own middleware, thus several solutions evolved in parallel without being compatible [145]. Therefore, accessing multiple Cloud platforms is challenging for the users.

In order to solve the interoperability problems, researchers are now working on Cloud federation, also known as *Sky Computing*. Sky Computing [111] is to Clouds what Computational Grids are to supercomputers on another scale. If Grid Computing is the aggregation of distributed heterogeneous resources, Sky Computing is the aggregation of distributed heterogeneous Clouds. The development of Sky computing will provide the transparent access between Clouds that the Grid brought between parallel machines.

In order to have different Clouds compatible together, standards are being developed [146]. However, the development of standards is long and difficult, therefore, while the standards are in maturation, users develop software compatible with multiple Cloud platforms [41]. These problems are the Cloud equivalent of interoperability problems between grid middleware, where each middleware has its own API, introduced in Section 1.2.

2.2 Scheduling in Parallel and Distributed Environments

The previous section presented the different architectures on which users can run parallel applications. Because platforms may be shared by multiple users and applications, applying scheduling strategies is necessary. Indeed, each application tries to use the resources without consideration for the others, therefore leading to a somewhat chaotic system. The hardware architectures presented in Section 2.1 are more or less hierarchical. Each architecture re-uses a simpler one and extends it to provide higher level of scaling. Larger platform size allow users to use more resources, however it also complicates the scheduling by adding more layers to manage.

This section introduces the scheduling problematics and present different levels of scheduling. Scheduling is already done by the operating system in a simple mono-processor machine to choose which program to execute. Scheduling of applications by the operating system is a research topic by itself so we do not look at this aspect. We are interested in the scheduling of multiple applications in parallel and distributed environments. Therefore, we look at job scheduling on clusters or parallel machines as well as on Grids in different contexts.

5. http://aws.amazon.com/ec2/hpc-applications/

2.2.1 Scheduling Basics

Job scheduling in parallel and distributed environment is a well known and studied problem in High Performance Computing [71, 79, 109, 172]. For simplicity, we use scheduling to describe both job scheduling and mapping. Thus, a scheduler is in charge of choosing where and in what order applications may be executed. The scheduling of applications may greatly change the performance of the system. Therefore it is important to take good scheduling decisions. However, while this description is simple, the problem of scheduling is known to be NP-complete in most cases [92]. Thus, most scheduling algorithms use heuristics.

In the field of HPC, different kind of applications are used. They can be sequential, using only one processor, as well as parallel, using several resources concurrently. Parallel applications can be rigid, moldable or malleable [82]:

Rigid applications have been designed to be executed on a fixed number of processors that can never change. A video player can have a thread dedicated to video decoding and one for audio. This type of applications also come from bad software design where the developer only designed the application for a particular environment.

Moldable applications can be executed with different number of processors, but once the execution started, this number can not change. Usual MPI programs take a list of resources as input parameter and are executed on these resources.

Malleable applications are the most permissive ones. The number of processing elements used by the application can be modified "on the fly" during its execution. This kind of applications is often an iterative application where the number of processors can be changed at the beginning of each iteration.

Tasks can sometimes be scheduled independently [158], but because of the complexity of most HPC applications, they are often represented as Directed Acyclic Graphs (DAGs) [130, 138]. In a DAG, nodes represent computing tasks and edges correspond to data dependency between the tasks. A correct schedule for a DAG is one that does not violate data dependencies between tasks. It can be obtained by executing the tasks in a topological order of the graph. Tasks composing a DAG can be simple sequential ones as well as complex malleable tasks [121], therefore obtaining parallel tasks that can be executed in parallel. This kind of parallelism is called mixed-parallelism [135].

Parallel applications executed in distributed environments are often simulations of some phenomenon. Because simulating a model is not 100% accurate, researchers often perform numerous identical simulations trying all possible, or a very large number of, input parameters. This method is referred as parameter sweep [64, 70]. All the tasks are identical from an abstract point of view, therefore it is possible to design special purpose scheduling heuristics for this kind of problem. If several parameter sweep applications are executed on the same platform at the same time we refer to them as bags of tasks [132, 171] where each bag contains an homogeneous set of tasks to execute.

All these different representation and grouping of tasks lead to different designs for scheduling heuristics. Of course, it is possible to mix all of the categories described above and have to schedule multiple bags of tasks represented by DAGs composed of sequential, rigid, moldable, and malleable tasks.

2.2.2 Online and Offline Scheduling

As we presented earlier, the scheduling can be done in different contexts. It is possible to try to schedule independent applications, bags of tasks, or DAGs. This categorization depends on the tasks to schedule themselves. Another possible categorization is the context in which the tasks are scheduled. If all the jobs to schedule are known at schedule time and all the information about the tasks is also known, we can apply sophisticated heuristics (optimal algorithms usually exist but are far too slow to be used in practice) that should give good results. However, if jobs arrive continuously, the scheduling algorithm does not know the future. Therefore, with less knowledge, results are expected to be less good.

Online scheduling [23] describes a scheduling context where jobs may arrive over time. This is usually the case in clusters or Grids where users submit independent jobs over time. Usual online scheduling algorithms schedule jobs as soon as they arrive so it takes only one job into account. The decision taken at this time is done using some heuristic, and even if the decision is the best (according to the heuristic) when taken, it may prove to be wrong when other jobs arrive later. Because the scheduler only treats one job at a time, it is usually very fast to take a scheduling decision for the job.

Offline scheduling [101] refers to a context where all information about the jobs is known prior to scheduling. The scheduler can thus take into account the arrival dates, the durations, or any other parameter. Therefore, the scheduler is able to produce the optimal schedule, which was not possible in an online context. This kind of scheduling problem arrives when a user submits a set of identified jobs, as it is the case when a bag of tasks is submitted for example. However, obtaining an optimal schedule in an offline context is very long, therefore, even in offline contexts, heuristics are used. However, these heuristics are slower than online heuristics because they have more information to take into account, but they can make supposedly better decisions.

It is possible to mix the two kind of schedulers. For example, in an online context where jobs arrive over time, it is possible to keep new jobs in a queue for some time and use an offline algorithm when enough jobs are present. This dynamic online scheduling tries to take advantage of the maximum knowledge each time to obtain better results. However, the scheduler must decide how long to wait before performing a schedule, or how many tasks to receive. Taking these decisions is also a very hard choice. If the scheduler holds the tasks too long, the additional waiting time will not be compensated by a better schedule. On the other hand, it is also possible to use an online heuristic when all the jobs are known in advance. This can be done to have a faster execution time for example.

2.2.3 Evaluation of a Schedule

Because heuristics are used to obtain schedules of jobs, it is necessary to study their performance by comparing the different results. The comparison is done on different measurements of the resulting schedule. Each measurement gives us one metric to perform the comparison. Usual scheduling heuristics try to improve one metric, but more advanced techniques can mix different metrics together and try to optimize a multi-criterion problem [72, 113]. Comparing different heuristics may be difficult because they can only be

compared in the same context in order to have sound results.

2.2.3.1 Real Execution vs. Simulation

In order to obtain sound results, comparisons must be done on the same set of tasks to be scheduled on the same platform. The only difference must be the heuristic used. Therefore, a common practice for researchers is to simulate the behavior of their heuristics with the help of a simulator. A simulator is a software attempting to mimic the behavior of a system by executing models of the reality. To change the behavior of a simulator, it is just necessary to change the model or the input parameters of a model, therefore the simulation environment is entirely controlled.

Reproducing results is not possible in a real distributed environment. There are always some uncontrolled parameters that may change the executions between two apriori identical experiments. Therefore, by using a simulator where every aspect is controlled, researchers are able to perform numerous simulations with the ability to reproduce results. In order to compare several scheduling heuristics, having a fully controlled environment make it possible to compare the heuristics in exactly the same conditions. Simulation results are only depending on the heuristics, without any external influence.

In computer science, it is not possible to stop users from submitting jobs in a production environment just to test a new scheduling heuristic. It is necessary to evaluate, optimize, and ensure the correctness of the heuristic before deploying it in the scheduler of the production environment. Researchers from other scientific fields have similar practical issues: It is not possible to make hundreds of planes crash to see if a new design of wings works properly. A reasonable approach is to validate the new design by performing simulations and only then test it in real life.

However, because simulations use models of reality, it is necessary to use a simulator with accurate models so that results are as close as possible to reality. Better models provide better results. However, they become more complicated and longer to simulate. Simulating a distributed environment where network contentions are not represented is faster than using a model taking it into account, however, the results may be very wrong if the communications are important for the simulations. The choice of the model is very important depending on the experiments that are done. We can also cite the example of a meteorological model that predicts the weather very accurately but that takes more time to simulate than just waiting to see the weather. Nobody would use this model because it is too slow. It is thus necessary to find a compromise between model complexity, execution time of the simulation, and accuracy.

2.2.3.2 Metrics

Scheduling heuristics are designed and often compared using simulators. Because of the reproductability of simulations, researchers can run a single experiment several times and compare the heuristics on different metrics. Depending on the context, some metrics may be relevant or not.

Makespan: The makespan [80] is probably the most used metric in job scheduling.

It corresponds to the total duration needed to execute all the jobs of a workload: it is the difference between the last job completion time and the first job start time. The makespan is used when all the jobs are known before giving the schedule in a offline context. For example, when scheduling a bag of tasks. It can also be used in online contexts where the last job is submitted long before the end of the execution. However, if the makespan depends on the last arrival date, there is no justification for its use. When scheduling heuristics are based on the makespan, they try to minimize it so that all the computations are finished as soon as possible. The makespan is not influenced by the order in which jobs are executed. Only the first job to start and the last job to finish are used.

Throughput: The throughput [80] of a system corresponds to the amount of work done during a period. When a continuous flow of jobs have to be executed, maximizing this metric will ensure that the maximum work is done at each step. Throughput is used extensively in steady-state scheduling where a continuous flow of identical jobs arrive [28]. It can also be used in an online context where the scheduling heuristic tries to maximize the throughput each time a new job arrives or finishes. Maximizing the throughput of a system will ensure that it was not possible to make more work during this period.

Stretch: The stretch [31] of an application is defined as the ratio of its time spent in the system over its duration in a dedicated mode, i.e., when it is the only application executed on the platform. Because the stretch only takes one job into account, the stretch can be optimized on both online and offline contexts. It is possible to minimize the average stretch so that the system executes jobs faster. Another usual objective is to minimize the maximum stretch, thereby enforcing a fair trade-off between all applications [32].

Energy: The energy consumption [170] is a metric aiming at diminishing the energy consumption of computations. This metric is always used in multi-criterion optimization problems, otherwise all the jobs would be scheduled sequentially on the most energy efficient machine. Studies have proposed scheduling heuristics optimizing makespan and energy and have obtained a similar makespan while consuming less electric power [25].

Scheduling time: Scheduling techniques are employed to improve some of the previous metrics. Therefore, the time taken by the scheduler to obtain a schedule should be taken into account when comparing heuristics. If the goal is to minimize the makespan of an application, it would be very bad if the time to obtain this schedule was so great that it would actually take longer that using a very simple, but fast, heuristic obtaining a bad schedule. Thus, the execution time and theoretical complexity of the heuristics is also a very important metric to consider during the design of scheduling heuristics.

2.2.4 Scheduling on Clusters

Multitasking operating systems are able to execute several tasks at once by allocating resources to the different tasks for very short period of time. This method is called *time sharing* because applications share the resources in turn. This strategy can be applied in clusters as well [95, 174]. However, using a time sharing policy with high performance parallel application generally leads to very poor performance. Indeed, switching between applications has a cost, therefore it is usually better to execute applications one after the other. Furthermore, two applications with large memory requirements may not be able

to run concurrently on a single machine of the cluster. For these reasons, schedulers in clusters usually use a *space sharing* policy where each application has a dedicated access to the resources for a given period.

Therefore, in order to execute a job on a cluster, users must submit their job to a batch scheduler [24, 81] that gives a dedicated access to the resources for some time. Batch schedulers keep a Gantt diagram of the resources. One axis of the diagram represents the machines and the other axis corresponds to time. When a job is submitted, the batch scheduler looks for a place where the application can be executed. Therefore, finding a schedule is equivalent to finding a tiling of this 2D plane, where the plane represents resources availability over time and tiles represent jobs. Thus, submissions of jobs must include a description of their requirements: the number of processors needed as well as their duration. Most schedulers are only able to schedule rigid tasks.

Giving the duration of a job before its execution is usually impossible, therefore a walltime is provided. The walltime of a job is the expected duration of the job. Batch schedulers use this information to perform the scheduling. If the walltime is underestimated, jobs are usually killed, so the users give an overestimation of the expected execution time.

Let us now list some of the most commonly used algorithms in batch scheduling:

First-Come First-Served (FCFS) [153] is the simplest scheduling algorithm used in batch scheduling. This algorithm schedules all incoming jobs at the end of the waiting queue. Figure 2.6 illustrates this process.

Whenever a job finishes before its walltime, FCFS compresses the scheduling of jobs present in the system. To compress the schedule, the jobs are resubmitted in their submission order and can use the newly available resources. Therefore, the jobs can only start earlier than decided on submission, but they can not be delayed by jobs arriving after them.

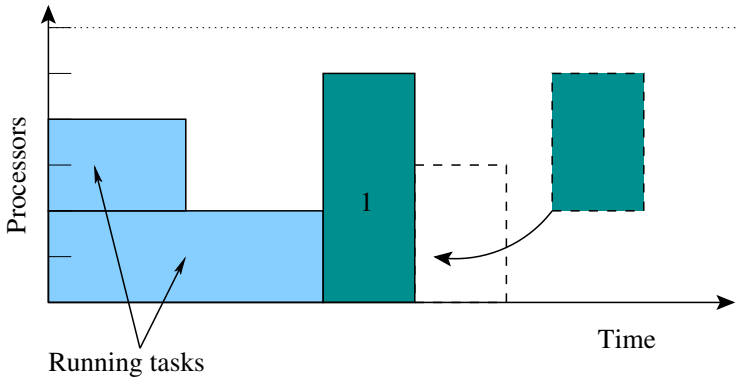


Figure 2.6 – First-Come First-Served example.

Conservative Backfilling (CBF) [129] tries to take advantage of empty spaces present in the waiting queue. Instead of just looking at the end of the queue for free processors, CBF looks in the whole queue and tries to schedule the job in an empty space (Backfilling). A job can only be backfilled if the schedule of jobs already present in the system is not modified (Conservative). Figure 2.7 illustrates the process of backfilling. The new submitted job passes in front of a job already present in the queue, but does not delay it.

When a job finishes earlier before its walltime, CBF compresses the rest of the schedule. To compress the schedule, jobs are resubmitted by increasing start time instead of using the submission order as in FCFS. Because of backfilling, if jobs were resubmitted in their submission order, some may be delayed [129]. Therefore, as for FCFS, jobs can never start later than when it was decided during the submission of the job.

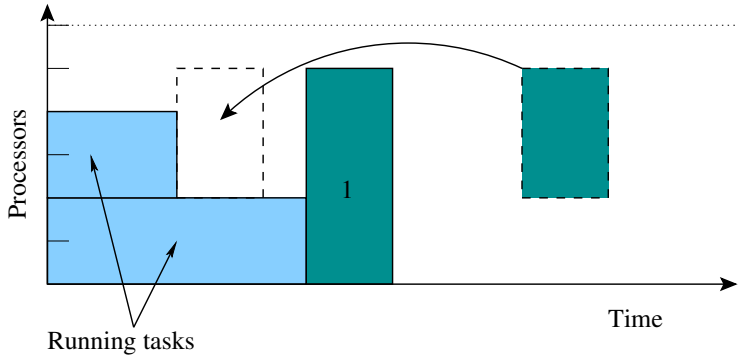


Figure 2.7 – Conservative Backfilling example.

Aggressive Backfilling (ABF) [78] is a variant of CBF. When a new job arrives, the algorithm looks for free processors in the queue as soon as possible. If the job can be scheduled on these processors, it is backfilled without regarding the consequences on the jobs already present in the queue (Aggressive). In Figure 2.8, job 3 can fit on the few resources left in front of the queue, therefore it is scheduled on them. However, because it has a long walltime, it delays the first job that was in the queue. Delaying a job may also cause other jobs to be delayed if there are jobs scheduling just after it.

Contrarily to FCFS and CBF, the aggressive backfilling can not give any guaranty on the start time of a job. Indeed, because any job can be delayed, the schedule is only sure not to be modified at the beginning of the queue when all resources are occupied. Furthermore, because any job can be delayed, some jobs may never be executed. Indeed, in some setups, arriving jobs would always jump in front of large old ones, delaying them indefinitely.

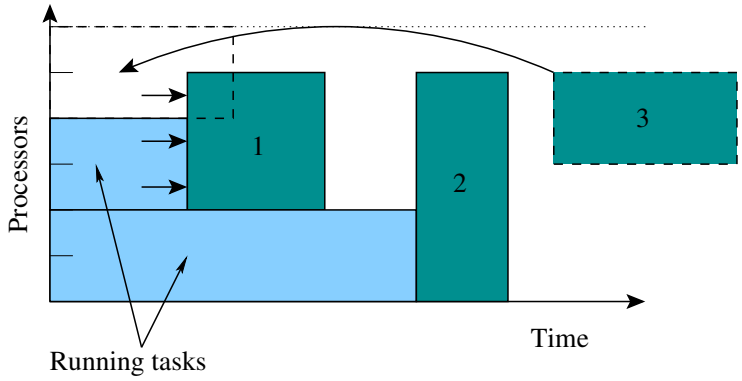


Figure 2.8 – Aggressive Backfilling example.

Easy Backfilling (EASY) [122] is an extension of the aggressive backfilling that deals

with the starvation issue. The algorithm is the same, except that the oldest job in the queue can not be delayed. As time passes, if a job is delayed for a long time, it will become the oldest job and will have a fixed starting date. In Figure 2.9, the new job could start execution as soon as submitted, however, the oldest job can not be delayed, therefore, the newly submitted job is scheduled after it, delaying another job.

While it prevents starvation inherent to ABF, EASY keeps the inability to ensure that a job will not be delayed. It is thus impossible to have an upper bound for the starting time of the jobs.

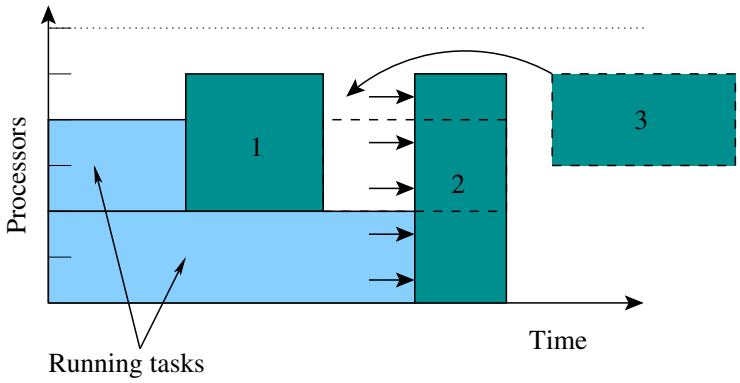


Figure 2.9 – Easy Backfilling example.

All the above algorithms work in an online manner. They schedule each job independently upon submission. Other algorithms work in an offline manner. For example, Backfilling with Lookahead [155] uses packing techniques to maximize the resources utilization at each job submission. The packing techniques use the knowledge of all the jobs each time. This kind of algorithm is not widely used because the execution time needed to execute the algorithm can easily become too long.

2.2.5 Scheduling on Grids

We defined Grids as the aggregation of distributed heterogeneous resources. Therefore, scheduling jobs in a Grid is quite different than scheduling them on a cluster. Indeed, clusters are usually homogeneous and have a reasonable size. However, a Grid can embed different kind of hardware, software, and scheduling policies. Furthermore, the number of resources available in a Grid is a lot larger than on clusters and the access is done differently. Clusters are accessed through resource management systems but Grids are usually accessed through a grid middleware. Job scheduling in a Grid must therefore take all these additional constraints into account.

Scheduling on Grids is usually called meta-scheduling [59, 106]. Instead of deciding when and on which processors to execute a task, the meta-scheduling here is to choose on which parallel resource should be executed a job. Using the term mapping would be more adequate because the meta-scheduler, or resource-broker, maps jobs onto parallel resources. Once the parallel resource is chosen, the job is scheduled on the processors by an underlying batch scheduler, therefore using a two-level scheduling [164]. The algorithms

used for mapping are online ones [152] because jobs arrive continuously, and the algorithms used on the parallel resources may be one of those presented in Section 2.2.4.

The selection of the parallel resource to execute independent jobs can be done using different online algorithms [37, 73]. Depending on the amount of information available to the scheduler, different heuristics can be applied. Indeed, a scheduler aware of the heterogeneity of resources as well as estimation of jobs execution time will take better decisions.

Meta-scheduling decisions can be taken using some of the algorithms listed here:

Random selects a resource randomly. It does not need any information about jobs or resources. However, it usually provides poor results. In a context where all jobs and resources are similar, it should provide an acceptable load balancing.

Round Robin (RR) select resources in one at a time and starts over when all resources have been selected. As for random, this algorithm requires no information about jobs or resources. In an homogeneous context, RR will make a good load balancing among the resources. However, in an heterogeneous context, some resources will get overloaded and other will be underused. Indeed, each resource will get the same amount of work but some will require more time to process their part of work.

Optimistic Load Balancing (OLB) assigns each task on the resource expected to be available first. The objective is to keep the platform as busy as possible. This algorithm only requires information about the resources state. Because it does not take the execution time into account, OLB can lead to poor results.

Minimum Execution Time (MET) assigns jobs on the resource where the job is expected have the smallest duration. This algorithm requires an estimation of the execution time of each job on each resource. Furthermore, if it is used in a context where tasks are characterized as consistent (a machine running a task faster will run all the tasks faster), it will assign each task to the same machine.

Minimum Completion Time (MCT) needs an estimation of the completion of the jobs of each resources and it assigns the job to the resource able to finish it first. It combines advantages of OLB and MET by assigning the task on the resource with the earliest date of completion. When possible, this heuristic should be used as it takes resource heterogeneity, platform load, and jobs' execution time into account.

Even if the meta-scheduler only addresses the selection of a parallel resource, it may still be a bottleneck when numerous submissions arrive. Therefore, researchers work on scalability issues, often by using a distributed meta-scheduler [33, 118]. The meta-scheduler can distribute the scheduling decisions among a set of computers. Connections can be done using a Peer-to-Peer topology [102], or by having more predictable topologies such as a hierarchy [56].

Working on a single Grid platform may not be sufficient, therefore it may be necessary to federate multiple Grids. In this context, the interoperability issues introduced in Section 1.2 must be taken into account during the scheduling phase [112]. Using several Grids at once adds yet another layer of scheduling called meta-brokering. At the top, the meta-broker selects a Grid, then the resource-broker of the Grid selects a parallel resource and finally, the physical resources are assigned by the local resource manager. Most solutions to tackle the problem of interoperability is to use a meta-broker coordinating the different Grids [147, 169].

2.3 Parallel and Distributed Computing in Practice

In the previous sections, we described hardware architectures and scheduling algorithms, used by the parallel and distributed computing community. In this section, we present real software and infrastructures deployed in practice.

We start by presenting some resource management systems deployed on production parallel machines or clusters. We then expand to real Grids and present research and production Grids. Then, we present some grid middleware used to manage jobs in Grids with a special focus on GridRPC based middleware. Finally, we present some simulation toolkits that are used in order to simulate these parallel and distributed platforms.

2.3.1 Batch Schedulers

A batch scheduler, or Local Resource Management System (LRMS), is a software that manages resources in a cluster. It is in charge of scheduling and mapping jobs onto resources. Different implementations are available, each one with its own set of commands to manage jobs preventing simple compatibility. Each batch scheduler can implement a different scheduling policy, including the ones described in Section 2.2.4. Some implementations are designed to be used in production environments and therefore focus on stability, and other have a research approach so they include a large set of advanced features.

LoadLeveler [108] is a proprietary batch scheduler developed by IBM. The scheduler is now part of the Tivoli Software [16] as the Tivoli Workload Scheduler. LoadLeveler has been used in production environments for more than 15 years. The scheduler is an entirely distributed program with extensive failover and self-repair capabilities to survive even severe system events, usually without administrator intervention. Furthermore, it offers jobs check-pointing and suspension. The software scalability is also a concern of the development team. The scheduling algorithm used in LoadLeveler uses back-filling to improve job management.

SLRUM [20] (Simple Linux Resource Management Utility) is an open-source resource manager. It is developed and supported by different companies and laboratories including IBM and Sun Microsystems. SLURM includes a scheduling facility as well as monitoring tools. The SLURM core can be extended using a plug-in mechanism. Dozens of plug-ins are distributed with SLURM to perform all kind of tasks, including managing resources limits, jobs prioritization, logging, economics, and many more. SLURM is used by more than 1000 large systems including the most powerful supercomputer in Europe, the Tera100 with 140,000 processing cores, or the BlueGene/L at Lawrence Livermore National Laboratory (LLNL) with 106,496 cores.

OAR [53] is an open-source versatile batch scheduler designed for large clusters. Its development is mainly done by Mescal team in the LIG (Laboratoire d'Informatique de Grenoble) computer science laboratory. OAR aims at providing cutting edge features. The scheduler uses a CBF policy to schedule jobs in different priority queues. In addition to normal queues, OAR offers a best-effort queue to maximize the usage of idle resources, but best-efforts jobs may be killed anytime when a submission on another queue is done. Other advanced features include moldable jobs management, admission rules, hold and resume

jobs, submissions inside an existing submission, advanced reservations, energy savings. OAR is a collection of different tools interacting with a MySQL database. Developing a new module, such as a new scheduler, can then be done in any language able to access the database. Furthermore, all the independent modules make it easier to maintain the software. OAR is used by in the GRID'5000 infrastructure to manage scheduling on the different clusters.

2.3.2 Experimental Testbeds

In this section, we depict different Grids and divide them into two categories. Production Grids are infrastructures aiming at providing all the necessary features for applications of scientists in different areas. Because production Grids are used for real experiments, the technologies used need to be mature. On the other hand, research Grids are used by computer scientists to develop and test new Grid technologies. Once new technologies have been extensively validated, they can be introduced in production Grids.

2.3.2.1 Production Grids

EGI (European Grid Infrastructure) [3], formerly EGEE (Enabling Grids for E-sciencE) until April 2010, is a project started in 2004 supported by the *European Commission*. It aims at providing researchers in academia or business with an access to a production level Grid Infrastructure. EGI has been developed around three main principles: *(i)* provide a secured and robust computing and storage grid platform; *(ii)* continuous improvement of software quality in order to provide reliable services to end-users; and *(iii)* attract users from both the scientific and industrial community. It currently provides around 150,000 cores spread on more than 260 sites in 55 countries, and also provides 28 petabytes of disk storage and 41 petabytes of long-term tape storage, to more than 14,000 users. Whereas the primary platform usage mainly focused on high energy physics and biomedical applications, there are now more than 15 application domains that are making use of the EGI platform.

TeraGrid [14] is an American project supported by the NSF since 2001. It is an open production and scientific Grid federating eleven partner sites to create an integrated and persistent computational resource. Sites are interconnected through a high speed dedicated 40Gb/s network. Available machines are heterogeneous, as one can find clusters of PCs, vectorial machines, parallel SMP machines or even supercomputers. The whole Grid has more than 2 petaflops of computing capability and more than 30 petabytes of online and archival data storage available to industrials and scientists. TeraGrid competes with the EGI infrastructure to be the world's largest, most comprehensive distributed cyberinfrastructure for open scientific research.

DEISA (Distributed European Infrastructure for Supercomputing Applications) [15] is a European project founded in 2002 by the European Commission. It is made up of a consortium of eleven leading national supercomputing centres from seven European countries. It supports pan-European research by providing and operating a distributed supercomputing environment all over Europe and aims at delivering a turnkey operational

solution for a future European high-performance computing system. The eleven centers are interconnected through a dedicated 10Gb/s network connection provided by GÉANT2 [5] and National Research and Education Networks. Computing resources include many kinds of computers and operating systems, all managed by a specific grid middleware to obtain a transparent access to all the resources.

2.3.2.2 Research Grids

GRID'5000 [34] is a French project started in 2003, supported by the French ministry of research, regional councils, INRIA and CNRS, whose goal is to provide an experimental testbed for research on Grid computing. It provides a nation wide platform, distributed on 9 sites (new sites are currently being added in Brazil and Luxembourg), containing more than 6,200 cores on 30 clusters. All sites are interconnected through 10Gb/s links, supported by the Renater Research and Educational Network [8]. As grids are complex environments, researchers needed an experimental platform to study the behavior of their algorithms, protocols, etc. The particularity of GRID'5000 is to provide a fully controllable platform, where all layers in the grid can be customized: from the network to the operating systems. It also provides an advanced metrology framework for measuring data transfer, CPU, memory and disk consumption, as well as power consumption. This is one of the most advanced research grids, and has served as a model and starting point for building other grids such as the American project FutureGrid.

FutureGrid [9] is a recent American project, started in October 2009, which shares the objectives of GRID'5000. Its goal is to provide a fully configurable platform to support Grid and Cloud researches. It contains about 5,400 cores present on 6 sites in the USA. One of the goals of the project is to understand the behavior and utility of Cloud computing approaches. The FutureGrid will form part of *National Science Foundation*'s (NSF) TeraGrid high-performance production grid, and extend its current capabilities by allowing access to the whole grid infrastructure's stack: networking, virtualization, software, and workflow orchestration tools. Full integration into the TeraGrid is anticipated by 1st October 2011.

OneLab [11] is a European project, currently in its second phase. The first phase, from September 2006 to August 2008, consisted in building an autonomous European testbed for research on the future Internet. The resulting platform is **PlanetLab Europe** [13]. In its second phase, until November 2010, the project aims at extending the infrastructure with new sorts of testbeds, including wireless (NITOS), and high precision measurements (ETOMIC) testbeds. It also aims at interconnecting PlanetLab Europe with other PlanetLab sites (Japan and USA), and other infrastructures. PlanetLab hosts many projects around *Peer-to-Peer* (P2P) systems. They rely on totally decentralized systems in which all basic entities perform the same task. Though initially outside the scope of grid computing, P2P has progressively gained a major place in grid researches.

2.3.3 Grid Middleware

A grid middleware is a software layer designed to hide the complexity and heterogeneity of a Grid. In this section, we present the basics on what is a middleware. In our work, we

developed features in a middleware relying on the GridRPC paradigm, therefore we detail the GridRPC approach to service execution and give some example of GridRPC compliant middleware.

2.3.3.1 Generalities on Grid Middleware

To hide the complexity of the Grid, a grid middleware must be able to provide a large variety of services. It must perform many tasks including transparent access to resources, job scheduling, job monitoring, platform monitoring, data management, security, *etc*. By using a middleware, software developers do not have to manage the placement and scheduling of their application by letting the middleware take the decisions.

Providing all these features is a hard task. Therefore, toolkits were developed to help in the development of services for the Grid. The Globus Toolkit [86] is the most famous and most used toolkit. It is developed by the Globus Alliance, an international collaboration that conducts research and development to create fundamental Grid technologies. The Globus Toolkit includes software for security, information infrastructure, resource management, data management, communication, fault detection, and portability. It is packaged as a set of components that can be used either independently or together to develop applications.

One of the most important grid middleware is gLite [117]. It was developed in the context of the EGEE project. The gLite distribution is an integrated set of components designed to enable resource sharing. It is designed to provide the necessary components to build a Grid. Each of the different components have a specific role. The services offered by the middleware can be grouped in five groups: Access Services, Security Services, Information and Monitoring Services, Data Services, and Job Management Services. This last service manages jobs scheduling and execution. GLite can be used in a multi-cluster environment, therefore it supports many batch schedulers. The gLite approach is very complete and provides all the necessary features, but it includes heavy mechanisms such as advanced security options. Therefore, other mechanisms have been developed.

2.3.3.2 The GridRPC Approach

A simple and efficient approach to provide transparent access to resources consists in using the classical *Remote Procedure Call* (RPC) method. This paradigm allows an object to trigger a call to a method of another object wherever this object is. The object can be a local object on the same machine, or a distant one. In this latter case the communication complexity is hidden with an abstraction layer. Different RPC standards exists. Among those, CORBA [137] and Java RMI [161] are the most used.

The classical RPC paradigm was extended in the context of Grid Computing. The GridRPC [154] approach was standardized by the Open Grid Forum in order to provide an efficient remote procedure call in a Grid context. Furthermore, the standardization enables interoperability between different middleware [163]. An application developed with the GridRPC paradigm should be able to use any GridRPC compliant middleware, thus enabling the application to run in different contexts. Grid middleware using this paradigm can also be referred to as Network Enabled Servers (NES) [127].

Because of the distributed environment offered by Grids, the GridRPC paradigm adds a mapping service to the simple RPC calls. The architecture of a GridRPC middleware is depicted in Figure 2.10. Three components are present. First, the server which provides services. The server registers the services to an agent, or registry. This second component is in charge of referencing the different services available in the middleware. The last component is the client. When the client needs to execute a service, it contacts the agent which returns one or more handles to contact the servers able to execute the request. Some scheduling can be performed at this stage to select the best servers according to some metric. With the server handle, the client can use a normal remote procedure call on the server and wait for the results.

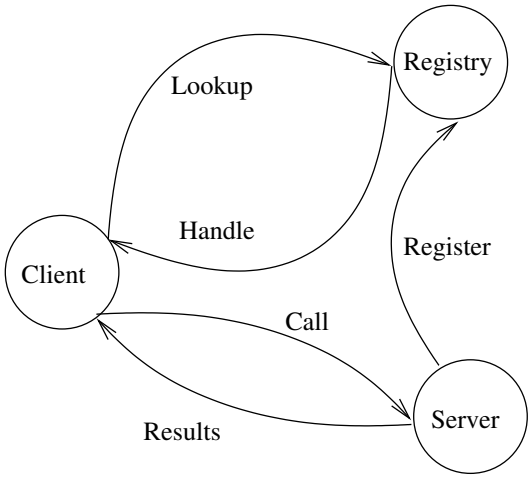


Figure 2.10 – The GridRPC architecture.

2.3.3.3 Examples of GridRPC Middleware

NetSolve/GridSolve [62] was one of the first grid middleware implementing the GridRPC approach in 1996. Its architecture includes the three components defined by the GridRPC standard. Once the agent is started, servers can register their services to the agent. Then, servers regularly send information about their state. Data include CPU usage, available memory, etc. The agent is centralized which is a bottleneck for performance. It is possible to duplicate the agent, but it is not really distributed because each instance of the agent needs the knowledge of the entire platform. The scheduling done by NetSolve returns a sorted list of servers to the client.

Ninf/Ninf-G [150] has an architecture very close to NetSolve. The functionalities are also similar. Ninf-G is developed using the Globus Toolkit. The agent is also centralized, however it is divided into several components with different roles, thus improving a little the scalability. The scheduler interrogates the performance database to take scheduling decisions and the database which updates its values by itself by calling a predictor.

DIET [54, 58] is another grid middleware based on GridRPC. Because we use it to validate our results, we present its architecture in details. Both NetSolve and Ninf use a

single agent. In order to distribute computations done by the agent, DIET extends it to a distributed hierarchical set of agents. Therefore, the scheduling can be distributed among the agents. A possible DIET hierarchy is presented in Figure 2.11. At the bottom of the hierarchy are the computational servers. They are hidden behind *Server Daemons* (SEDs). A SED encapsulates a computational server, typically on a single computer, or on the gateway of a cluster. A SED implements a list of available services. A SED also provides performance prediction metrics sent along with the reply whenever a request arrives. These metrics are the building blocks for scheduling policies. The SEDs services are exposed to their parents: the *Local Agents*. An agent has essentially two roles. First it forwards down incoming requests, and then it aggregates the replies and does partial scheduling based on some scheduling policy (shortest completion time first, round-robin, heterogeneous earliest finish time, ...). At the top, the *Master Agent* (MA) is the entry point of the hierarchy. Every incoming request has to flow through the MA. Clients can contact the MA via the CORBA naming service. The MA relies on the tree of local agents to forward the requests down the hierarchy and distribute the scheduling among them.

Because the MA is the only entry point of the hierarchy, it could become a bottleneck. Thus, to tackle this problem, several DIET hierarchies can be deployed alongside, and interconnected in a Peer-to-Peer fashion using CORBA connections.

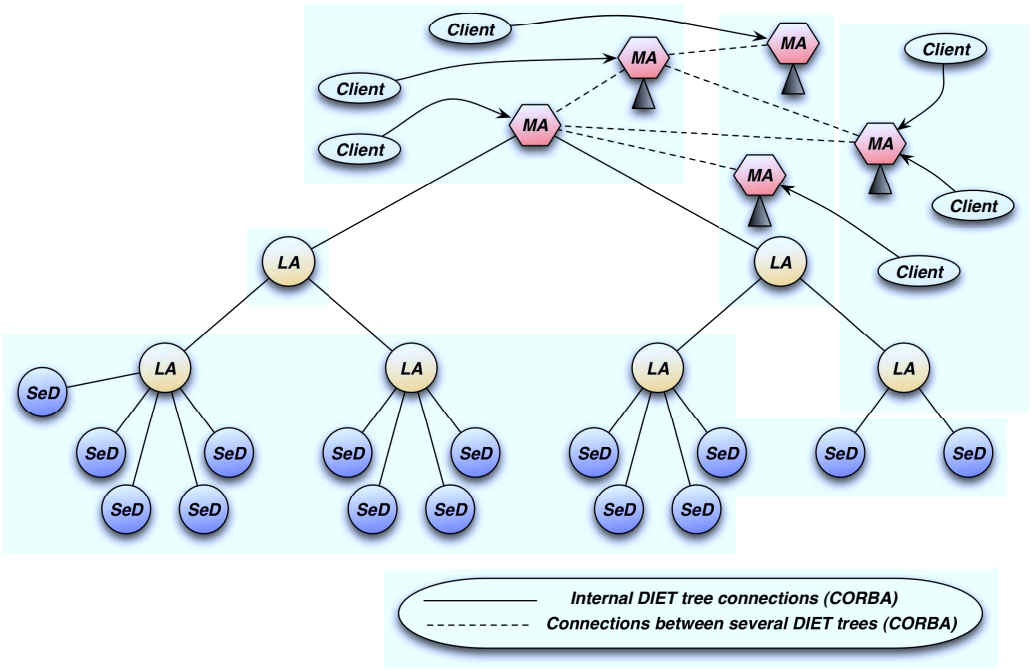


Figure 2.11 – The DIET middleware architecture.

2.3.4 Simulation Toolkits

Researchers often use simulations to develop and validate new algorithms. Some use home made simulators that can only be used for one task, but others use generic simulation toolkits designed to simulate distributed environments. Developing a fast, accurate, and generic enough simulator is a very hard task, so the best solution is to use an existing toolkit providing all the needed features. There are numerous simulators each one with a specific area. For example, in networking, NS2 [10] is the most used packet level simulator, but there are others such as GTNetS [144].

In the area of distributed computing, many simulation toolkits exist. Bricks [162] focuses on simulating client-server architectures. ChicSim [142] and OptorSim [29] are specifically designed to study data replication on Grids. PlanetSim [91] and PeerSim [107] are for the simulation of P2P applications. These toolkits have specific objectives that are out of the scope of our work, therefore we do not detail them. This section present more details on simulation toolkits most related to our work. We present toolkits designed to simulate distributed environment and batch schedulers.

GridSim [40] allows modeling and simulation of the different entities in parallel and distributed computing systems. It can model users, virtual organizations, applications, resources, and schedulers for design and evaluation of new scheduling algorithms. The toolkit concentrates on providing all the necessary abstractions to easily create a grid simulator. A resource can be a single processor or multi-processor with shared or distributed memory. The processing nodes within a resource can be heterogeneous in terms of processing capability, configuration, and availability. The resource brokers use scheduling algorithms or policies for mapping jobs to resources to optimize system or user objectives depending on their goals. While GridSim provides all the necessary abstractions to simulate a Grid, it only provide simple models for simulations. Basic models must be extended to perform simulations with complex requirements.

Alea Simulator [114] is based on the GridSim simulation toolkit. Alea extends the GridSim toolkit to provide all the necessary features to simulate batch schedulers present in parallel machines. Alea includes different queuing scheduling algorithms such as First-Come First-Served, Earliest Deadline First, and Easy Backfilling. The simulator also includes a centralized Grid scheduler to allow meta-scheduling and co-allocation of resources. Alea can take as input different standard job format such as the Grid Workload Format (GWF) of the Standard Workload Format (SWF). The simulator can produce different statistics on the workloads, the executions in batch schedulers, and compute different metrics. In order to tackle the problem of the models in GridSim, Alea implements its owns to suit their requirements.

SimGrid [63] is a discrete event simulation toolkit. It provides core functionalities for the simulation of distributed applications in heterogeneous distributed environments. The main goal of SimGrid is to facilitate the research in the area of parallel and distributed large scale systems such as Grids, P2P systems and Clouds. SimGrid relies on a scalable and extensible simulation engine. The simulation kernel of SimGrid relies on macroscopic models for computation resources. For network resources, SimGrid uses analytical models of TCP where communications flows are represented as streams in pipes. This enables faster sim-

ulations while conserving a reasonable level of accuracy [168]. When needed the GTNetS packet simulator can be used to obtain the maximum accuracy. SimGrid embeds different user APIs to tackle different needs such as simulation of client-server architectures, Grids, MPI applications, DAG scheduling, message passing, *etc*.

Simbatch [52] is a batch simulator built on top of SimGrid, therefore it relies on validated models for communication costs between hosts. Simbatch aims at providing a simple way of testing new batch scheduling algorithms and evaluate them by simulation. Another objective of the library is to provide results very fast so that it can be used as a prediction tool used by a grid middleware in real platforms. The simulator incorporates several scheduling algorithms including FCFS and Conservative Backfilling described in Section 2.2.4. Simbatch was extensively validated against the OAR batch scheduler and results show that the usual error on jobs starting time is 0.5% when jobs do not have input/output data and 1% with communications before and after. Simbatch can be used in any SimGrid application.

GridSim and Alea are two of the major simulation toolkits in distributed environments. They provide necessary abstractions to simulate platforms such as Grids. However, some of the models they use are simplistic. On the other hand, SimGrid and Simbatch have been intensively tested against real platforms to validate their models. Moreover, they are especially optimized to be very fast to execute. We use SimGrid and Simbatch in our work for these reasons.

CHAPTER 3

Tasks Reallocation in a Grid Environment

Contents

In this chapter, we present and evaluate the benefits of reallocating waiting jobs between clusters inside of a Grid. To measure the benefits of reallocation, we propose a generic solution based on the GridRPC standard. Then, we present a simulator implementing this generic solution to evaluate the gains brought by reallocation.

In order to study the gains of reallocating waiting jobs, we propose two reallocation mechanisms, each one using different scheduling heuristics. Using workload traces from real platforms, we evaluate the reallocation algorithms in different contexts: first, we concentrate our study on rigid tasks, and in a second time, we broaden our focus to moldable tasks.

3.1 Introduction

Grids are the aggregation of heterogeneous resources managed and shared by different institutions (see Section 2.1.4). A common type of Grid is the multi-cluster grid. Several parallel machines or clusters are interconnected through a high bandwidth network. Usually, clusters and parallel machines are managed by a local resource manager, also called batch scheduler. Each site may use a different scheduling policy and each resource manager is independent. As presented in Section 2.2.4, the schedulers require a number of processors as well as a walltime in order to make scheduling decisions and give the jobs a dedicated access to the resources. Therefore, the scheduling decisions taken by the batch scheduler are based on estimations and therefore are inherently wrong. Thus a mechanism dealing with the errors is necessary.

In most local resource management systems, when the walltime is reached, the job is killed, so users tend to over-evaluate the walltime to be sure that their job finishes its execution. Each time a job finishes before its walltime, the scheduling is modified and the jobs rescheduled. Rescheduling events triggered because of an early completion impacts the local scheduling, and thus impacts the global performance of the platform. Indeed, errors made at the local resource level may have a great impact on the global scheduling as shown in [30]. Moreover, errors on the scheduling are amplified by bursts of submissions as shown in [156]. Indeed, because of the numerous submissions, errors on walltime are accumulated and increases the loss of performance.

To connect the multiple sites of a multi-cluster Grid, using a Grid middleware is the simplest way. Users request to execute a service provided by the middleware which must give a performance prediction in order to provide the batch scheduler with the necessary parameters such as the walltime and the number of processors. The performance prediction can not be accurate, so the actual runtime of the service will be different from the walltime of the job submitted to the batch scheduler. Therefore, the middleware should be able to deal with the scheduling changes from each of the sites and propagate it to the entire platform. This propagation requires a mechanism able to move jobs from one site to another. We call this mechanism reallocation because we only consider moving jobs that have not started yet. On the other hand, migration [36] moves jobs on the fly while they are running. It is a common technique used in Cloud computing with the migration of virtual machines.

The current chapter focuses on the reallocation problematic. In order to give a practical solution to this problem, our main objective is to propose a generic solution at the middleware level so that it can be deployed on existing infrastructures. We aim at proposing a solution easy to implement within existing middleware. The generic architecture as well as the basic reallocation scheme were presented in [45, 66]. Our second objective is to propose different reallocation mechanisms, compare them, and decide, first if reallocation is really necessary, and then, which reallocation algorithm is the best, and finally, quantify the gains that users can expect. Middleware services are often parallel applications. As presented in Section 2.2.1 parallel applications can be rigid or moldable. Therefore, we evaluate the reallocation mechanisms on both types of tasks to have more general results. Reallocation of rigid tasks was presented in [48, 49], and on moldable tasks in [51, 50].

The rest of the chapter is as follows. In Section 3.2, we present some related works

on dynamic scheduling in Grids. In Section 3.3 we describe the architecture, mechanisms and the scheduling algorithms used in this work to perform reallocation. Then we describe the experimental framework in Section 3.4, giving information about the simulator we developed, on the platforms simulated with real-world traces, scenarios of experiments that were conducted as well as the metrics on which simulations are evaluated. Then, we present results on two different kind of tasks. First, we study reallocation of rigid tasks in Section 3.5, and then, we study the reallocation of moldable tasks in Section 3.6. Finally we summarize the chapter in Section 3.7.

3.2 Related Work on Dynamic Scheduling in Computational Grids

Guim and Corbalán [99] present a study of different meta-scheduling policies, including Less-JobWaitTime, Less-JobsInQueue, Less-WorkLeft and Less-SubmittedJobs, where each task uses its own meta-scheduler to be mapped on a parallel resource. Once submitted, a task is managed by the local scheduler and is never reallocated. In order to take advantage of the multi-site environment considered in our work, we use a central meta-scheduler to select a cluster for each incoming task because we place ourselves in the GridRPC context where clients do not know the computing resources. Also, once a task is submitted to the local scheduler, our approach let us cancel it and resubmit it elsewhere.

Sonmez et al. [156] present a method to diminish the errors made during a jobs burst in a multi-cluster environment. The method used consists in submitting the same job to several clusters (from 2 to all clusters) and when a job starts, all the other copies are canceled. To select the clusters, they use different heuristics such as MCT, Load Balancing, and Fastest Processor First. This method provides good results but adds an important load to the local resources management systems. Their approach is close to ours because it is also a middleware on top of an existing architecture. They use the multiple submissions to diminish the job response time while we use the reallocation mechanism. Our technique keeps the local resources management system less loaded because each job is submitted only once, but it needs more communications. With the multiple submissions, the first job starting sends cancellation messages to the other, so this technique is not well suited for heterogeneous platforms where a job starting later can finish earlier. Also, it requires synchronization between sites and a way to select which job to keep if two submissions start at the same time.

In order to migrate waiting jobs from one cluster to another, Yue presents the Grid-Backfilling in [173]. Each cluster sends a snapshot of its state to a central scheduler at fixed intervals. Then the central scheduler tries to back-fill jobs in the queue of other clusters. The computation done by the central scheduler is enormous since it works with the Gantt chart of all sites. All clusters need to be homogeneous in power because their scheduler is not able to adapt the walltime depending on the cluster speed. The study considers that jobs are submitted locally, so between two rescheduling phases, a cluster may be over used and others empty. Because jobs may be moved across sites, using a meta-scheduler upon jobs arrival would solve this issue. In our study, we use such a meta-scheduler, and we use

very simple queries to batch schedulers in order to let as few computations on the central scheduler during the reallocation phases.

Authors in [103] present a study of the benefits of using moldable jobs in an heterogeneous computational grid. In this paper, the authors show that using a Grid meta-scheduler to choose on which site to execute a job coupled with local resource management schedulers able to cope with the moldability of jobs improves the average response time. In our work, instead of letting the local schedulers decide of the number of processors for a job, we keep existing infrastructure and software, and we add a middleware layer that takes the moldability into account. Thus, our architecture can be deployed in existing Grids without modifications of the infrastructure already in place. Furthermore, this middleware layer renders reallocation between sites possible.

Our work is inspired by most of the works presented in this section. Our solution is placed at the middleware level and can be deployed on existing architectures. Because our jobs are submitted through the middleware, we can provide simple meta-scheduling as well as reallocation. Furthermore, the requests made to the middleware can be moldable and executed on a different number of processors depending on the current platform load.

3.3 Task Reallocation

In this section, we describe the proposed tasks reallocation mechanism. First, we present the generic architecture of the software (Section 3.3.1). Then we present the different algorithms used for the tasks reallocation (Section 3.3.2). Finally, we give some directions to implement our solution in the DIET grid middleware in Section 3.3.3.

3.3.1 Architecture of the Solution

We use an architecture similar to the GridRPC standard described in Section 2.3.3.2, which is implemented in middleware such as DIET or Ninf (see Section 2.3.3.3 for details). In a GridRPC middleware, clients query services present on the platform and launched by servers deployed on the computing resources. Therefore, the middleware takes the heterogeneity of hardware and software into account and requests can be executed anywhere transparently for the user. Because such a middleware is deployed on existing resources and has limited possibilities of action on the local resource managers, the mechanism we propose only uses simple queries such as submission, cancellation, and estimation of the completion time.

The architecture relies on three main components: the **client** has computing requests to execute, and contacts an **agent** in order to obtain the reference of a **server** able to process the request. In our proposed architecture, one server is deployed on the front-end of each parallel resource, in which case it is in charge of interacting with the batch scheduler to perform the submission, cancellation or estimation of the completion date of a job. The server is also in charge of deciding some parameters of the job such as the walltime and the number of processors allocated to the job. Benefiting from servers estimations, the agent maps every incoming requests using a MCT strategy (Minimum Completion Time [126]), and decides of the reallocation with a second scheduling heuristic.

The process of submission of a job is depicted in Figure 3.1. 1) When a client wants to execute a request, it contacts the agent. 2) The agent then contacts each server where the service is available. 3) Each server able to execute the request computes an estimation of the completion time and 4) sends it back to the agent. 5) The agent sends the best server to the client which then 6) submits its request to the chosen server. 7) Finally, the server submits the task to the batch scheduler of the cluster. 8) When the agent orders a server to reallocate a task, the latter submits it to the other server provided by the agent.

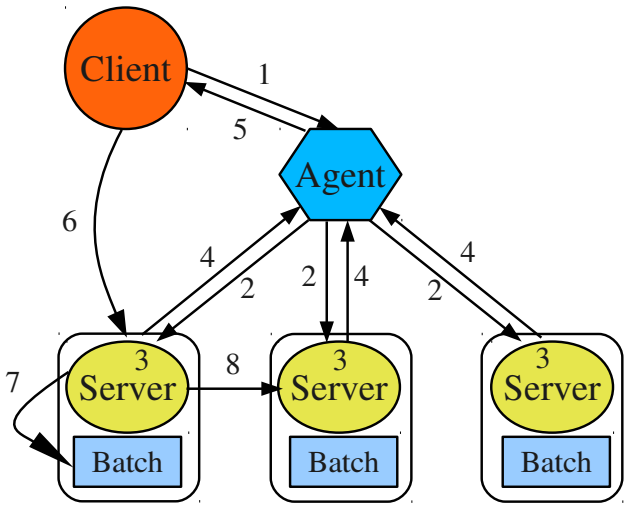


Figure 3.1 – Architecture of the middleware layer for reallocation.

3.3.2 Algorithms

This section presents the two versions of the reallocation mechanism (Section 3.3.2.1), and the scheduling heuristics used to provide reallocation (Section 3.3.2.2).

3.3.2.1 Reallocation Algorithms

The first algorithm, **regular**, presented in Algorithm 1, works as follows: It gathers the list of all jobs in the waiting queues of all clusters; it selects a job with a scheduling heuristic; if it is possible to submit the job somewhere else with a better estimated completion time (ECT) of at least one minute, it submits it on the other cluster and cancels the job at its current location; finally, it starts again with the remaining jobs. Submitting first and canceling later provides us with the guaranty that the job was reallocated properly.

The one minute threshold has been introduced to consider some data transfer that can take place, and to decrease the number of reallocations bringing almost no improvement. In the case where the platform network characteristics are known, the minute may be replaced by real estimations of transfer time.

To have a better idea of the purpose of the algorithm, consider an example of two batch systems with different loads (see Figure 3.2). At time t, task f finishes before its walltime,

Algorithm 1 Reallocation algorithm: "regular"

$l \Leftarrow$ waiting jobs on all clusters
while $l \neq \emptyset$ **do**
  Select a job j in l
  Get a newECT for j
  if $j.newECT + 60 < j.currentECT$ **then**
    Submit j to the new cluster
    Cancel j on its current cluster
  $l = l \setminus \{j\}$

thus releasing resources. Task j is then scheduled earlier by the local batch scheduler. When a reallocation event is triggered by the meta-scheduler at t_1, it reallocates tasks h and i to the second batch system because their expected completion time is better there. To reallocate the tasks, each one is sequentially submitted to the second batch and then canceled on the first one. In this example, the two clusters are identical so the tasks have the same execution time on both clusters, and the tuning of the parallel jobs (choice of number of processors to allocate to task h and i) is the same due to the same load condition. In an heterogeneous context, the duration and number of processors of the tasks would change between the clusters.

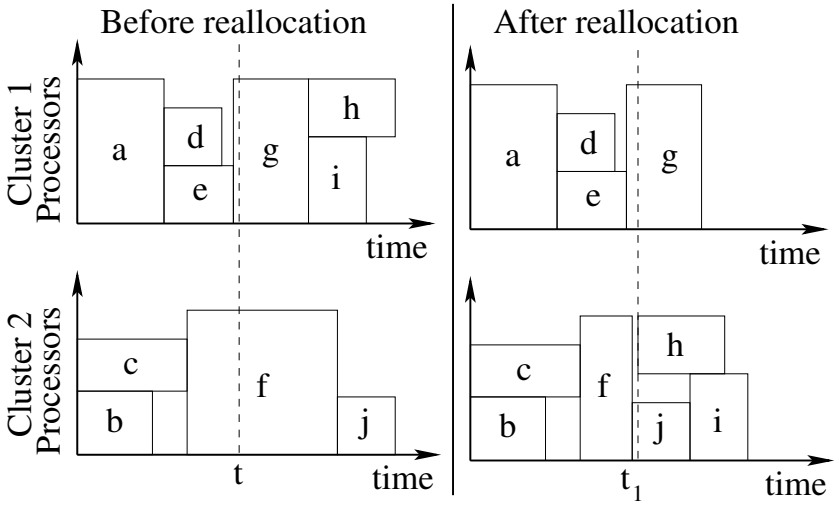


Figure 3.2 – Example of reallocation between two clusters.

The second algorithm, **all-cancellation**, detailed in Algorithm 2 starts by canceling all waiting jobs of all clusters. The agent keeps a reference for all jobs. Then, it selects a job with a scheduling heuristic. Finally, it submits the job to the cluster giving the minimum estimated completion time and loops on each of the remaining jobs.

Because the all-cancellation algorithm cancels and resubmits jobs, it is possible for it to cause starvation. The same job can always be selected last and thus always resubmitted at the end of a waiting queue. This might happen in a very loaded platform, or if the

Algorithm 2 Reallocation algorithm: "all-cancellation"

```
l ⇐ waiting jobs on all clusters
Cancel each job in l
while l ≠ ∅ do
    Select a job j in l
    Submit j to the cluster with MCT
    l = l \ {j}
```

scheduling heuristic makes bad choices. However, in all the traces we use for our experiments, there are always periods with little submissions of jobs so queues can be emptied, and all jobs executed without too much delay. The calm periods are often corresponding to the night or weekend or holidays in real life. Furthermore, forbidding the cancellation of the oldest job will prevent starvation of happening. Jobs may be delayed greatly but they will be executed eventually.

The reallocation event in both versions of the algorithm is triggered periodically every hour, based on previous works conducted in [45] where a smaller period did not change the results but required more network transfers and potentially more reallocations. The one hour frequency is enough because most jobs have a long duration, superior to one hour.

Because both reallocation algorithm use an estimation of the completion time, it is mandatory that clusters use a batch scheduling algorithm able to give some guaranties on the completion time to guaranty the results. The two main algorithms offering these guaranties are First-Come-First-Served (FCFS) and Conservative Back-Filling (CBF). Both algorithm make reservations for each job and jobs can never be delayed once the reservation done. However, jobs can be scheduled earlier if new resources become available. Batch schedulers using one of these algorithms are common. Other algorithms such as Easy Back-Filling (EBF) or the well-known Shortest Job First (SJF) do not guaranty a completion time and thus should not be used without adding specialized prediction mechanisms to the servers. Such specializations are out of the scope of this work and thus not discussed in this thesis.

3.3.2.2 Scheduling Heuristics for Reallocation

To choose the job that will be selected for reallocation, several heuristics are used: one online heuristic and five offline heuristics (see Section 2.2.2). The online heuristic takes jobs, one after another, while the offline heuristics are executed on a set of jobs. Because offline heuristics are more complex, they are supposed to give better results, but their execution time is bigger. The heuristics we compare are the following:

MCT Online algorithm. Assigns a task to the cluster that gives the minimum expected completion time. MCT takes jobs sequentially in their submission order.

MinMin/MaxMin Offline algorithms. Ask the expected completion time of all tasks and selects the one with the minimum/maximum value. These heuristics try to give priority to respectively small/large tasks.

MaxGain Offline algorithm. It gets the minimum expected completion time of each task. Then it computes the gain of moving each task. The gain is the time in seconds that the task would gain if it is reallocated ($Gain = CurrentECT - NewECT$). The task with the highest gain is selected and the heuristic starts again on the remaining tasks.

MaxRelGain Offline algorithm. Same as MaxGain, but divides the gain by the number of processors of each task, thus preferring small tasks, except if a large task has a very large gain.

Sufferage Offline algorithm. It gets the two estimated completion times for each task on each cluster, computes the sufferage value as the difference between the two best estimated completion times, and selects the task with the maximum sufferage value.

Concerning the execution time for each heuristic during the rescheduling event, MCT is the fastest. It takes the jobs in their arrival order without concern of the other jobs and it is executed n times, with n the number of waiting jobs. The offline heuristics on the other hand need to update information of all the remaining jobs each time a reallocation is performed, so the execution time may grow rapidly. In the worst case, if each task is reallocated, the number of estimations is n^2.

Notation: We have six scheduling heuristics, MCT, MinMin, MaxMin, MaxGain, MaxRelGain, and Sufferage, as well as two reallocation algorithms, namely regular and all-cancellation. Thus, we have 12 couples of algorithms that we refer in the remainder of this thesis as the junction between the name of the scheduling heuristic and "reg" or "can" for the reallocation algorithm. For example, to refer to the couple using all-cancellation with MaxRelGain, we use *MaxRelGain-can*. *MCT-reg* is the simplest reallocation mechanism corresponding to regular with MCT.

3.3.3 Implementation Design in DIET

The architecture of our solution to tackle the problem of tasks reallocation is based on the GridRPC standard. Therefore, the integration in a GridRPC compliant middleware should be facilitated. In this section, we give some directions to follow to implement tasks reallocation in DIET (presented in section 2.3.3.3).

To submit a job and benefit from the reallocation mechanism, the client makes a regular submission through the middleware. Indeed, the mechanism is transparent for the user. To manage the reallocation, we propose to use a dedicated entity. The reallocation entity is not an agent itself because it does not receive any requests from the client. It is executed in parallel of the DIET hierarchy and manages the requests reallocation. Therefore, in the following, we depict the implementation directions for this reallocation entity and the developments needed within the SEDs.

The architecture of the reallocation mechanism in DIET is the same as the one presented in Figure 3.1. The reallocation mechanism only manages requests already in the middleware, therefore it relies on the steps 2, 3, 4, 7, and 8 of the figure. The agent in the figure corresponds to the reallocation entity and the servers correspond to SEDs. In the

following, we will detail what is done at each step and how to implement it efficiently in DIET.

3.3.3.1 Implementation of the Reallocation Entity

The Master Agent (MA) of a DIET hierarchy knows all the available services and the way to access them. Therefore, the reallocation entity must be implemented at its level. We do not want to implement the reallocation mechanism within the MA to stay efficient for clients submissions scheduling. The problem of scheduling requests with knowledge of the whole hierarchy was already tackled in the context of DAGs scheduling through the MA_{DAG}.

The MA_{DAG} [35] is able to schedule multiple DAGs. This extension of the MA makes all the complex scheduling decisions for DAGs thus letting the MA concentrate on independent tasks. The MA_{DAG} is not able to send requests to the SEDs. It is just able to send SED references to the client that will submit the jobs himself. Indeed, to make the submissions all data from multiple DAGs would be stored in a single place, thus creating a bottleneck. However, the MA_{DAG} implements already many features that can be used in our reallocation entity, therefore, some of the code of the MA_{DAG} can be reused.

In addition to the knowledge of the platform, the reallocation entity will need to call services present on the servers. Indeed, the reallocation mechanisms described in Section 3.3.2 are based on internal services that have to be implemented in SEDs. These services offered by the SEDs are:

- *get_waiting_jobs()*: return the list of waiting jobs on the SED;
- *get_and_cancel_waiting_jobs()*: cancel all waiting jobs in the queue and return the list of jobs;
- *get_estimated_completion_time(job)*: return an estimation of the completion time for a job;
- *reallocate_job(job, destination)*: sends the job to the destination.

3.3.3.2 Implementation on the Server Side

DIET provides a special SED called SED_{Batch} [54]. This SED is designed to interact automatically with different batch schedulers, such as LoadLeveler, Torque, and OAR, without external intervention. The SED is able to take an incoming request and submit it to the batch scheduler. Submission to the batch scheduler is represented by step 7 in Figure 3.1. Internal mechanisms to cancel a job on a batch scheduler exist but no work has been done yet to use that kind of functionality. Therefore, almost all the simple interactions with batch schedulers required by our solution are already implemented in the middleware.

In order to take the decision of reallocating a job or not, the reallocation entity requires an estimation of completion time by calling *get_estimated_completion_time(job)*. The service is called in step 2 of Figure 3.1. Then, at step 3, the SED computes the estimation and step 4 corresponds to the answer to the caller. DIET can be used with CORI [55], a collector of information about the resources of the platform. CORI has a module named CORI_BATCH dedicated to obtain information about batch schedulers. This module can

obtain the estimation of the completion date of a rigid job. However, the module is not yet capable to manage moldable jobs. Therefore, it should be extended to be able to choose the number of processors for a request and update the walltime accordingly.

To get an estimation of the completion time, CORI_BATCH can query the batch scheduler (if the estimation feature is present) or use Simbatch (see Section 2.3.4) to simulate the batch system with its current load. The glue between Simbatch and CORI is not done yet. It is necessary to add a system call to a Simbatch application and read the result of the simulations. The binary to execute has already been developed. Afterward, some more work has to be done on the deployment of such a solution.

SEDs must implement the services *get_waiting_jobs()* and *get_and_cancel_waiting_jobs()*. The call and the answer to these services are represented by steps 2 and 4 in Figure 3.1. The $\text{SED}_{\text{Batch}}$ already implements a monitoring function to know the waiting jobs in the queue of the batch. The answer of the services contains the list of DIET IDs of jobs. Thus, to implement these services we can reuse existing features. If the all-cancellation algorithm is used, the call to this service should cancel the jobs as well as returning the list. To cancel jobs, the service will call the generic cancellation mechanism described above.

The last functionality required by our solution is the submission from one SED to another, done inside the call to *reallocate_job(job, destination)* represented by step 8 in Figure 3.1. The SED with the request to reallocate acts as a client, and can use the internals of the client API. On the other side, the SED receiving the request does not need to know if the request comes from a real client or from another SED, therefore no special treatment must be applied on the request. This service should be able to resubmit on the current SED. In the case of all-cancellation, a job may be resubmitted to SED where it already was. When the reallocation is done between two different SEDs, the SED will submit the job to the other SED and cancel it locally. The submission to the other SED may fail for unknown reasons. Therefore, in case of submission error during reallocation, the job should not be canceled locally.

Remarks:

- Input and output data may be associated with a request. When a request is reallocated to another SED, the input data should also be send. DIET already provides all the necessary data management thanks to the DAGDA data manager [57]. When a job is resubmitted from one SED to another, the associated data will be automatically migrated.
- Reallocation from one SED to another requires for the SEDs to be servers as well as clients. At the moment, this is not easily feasible. Indeed, because of library dependencies, a SED can not be linked to the client library. However, the DIET internal code structure is being redesigned and improved. This new version will be shipped next year with DIET 3.0. It will enable the link of a SED to the client library.
  DIET already provides most of the features required for task reallocation. Therefore, once the new version is available, the implementation should not require extensive efforts.

3.4 Experimental Framework

In this section we depict the experimental framework by presenting the simulator we implemented to run our experiments (Section 3.4.1), the description of the jobs (Section 3.4.2), the simulated platforms (Section 3.4.3), and the metrics used to compare the heuristics (Section 3.4.4).

3.4.1 Simulator

The simulator is divided using the same components as the ones in the GridRPC standard introduced in Section 2.3.3:

The **client** requests the system for a service execution. It contacts the meta-scheduler that will answer with the reference of a server providing the desired service. Then, the client can ask the server to execute its request.

The **meta-scheduler** matches incoming requests to a server using a MCT policy and periodically reallocates jobs in waiting queues on the platform using one of the reallocation mechanism and scheduling heuristic described in Section 3.3.2.

The **server** is running on the front-end of a cluster and interacts with the batch system. It receives requests from the client and submits jobs to the batch scheduler to execute the requests. Once submitted, a job can be canceled if its execution has not started yet. The server is also able to provide an estimation of the completion time for a request to the meta-scheduler. The estimation function finds the best number of processors for the job when dealing with moldable tasks and adjusts the duration of the task in accordance to the speed of the cluster. The estimation function is used at submission time and during the reallocation phases. In order to be able to run the reallocation algorithms, the server is also able to send the list of waiting jobs to the meta-scheduler.

The simulator is implemented in C with the SimGrid and Simbatch toolkits described in Section 2.3.4. We use these simulation toolkits because they provide accurate and efficient models that enables us to simulate the environment we need. Each component of the simulator is represented by a SimGrid process and all inter-process communications are done using the message passing provided in the toolkit. The server component communicates with Simbatch and uses this tool to simulate the presence of a real batch system. The SimGrid layer represents the middleware while the Simbatch layer corresponds to the batch systems that would be present in a real environment.

3.4.2 Jobs Traces

We built seven scenarios of jobs submission, where for six of them, jobs come from traces of different clusters on GRID'5000 for the first six months of 2008 (Table 3.1 gives the number of jobs per month on each cluster as well as the total number of jobs); The seventh scenario is a six month long simulation using two traces from the Parallel Workload Archive [12] (CTC and SDSC) and the trace of Bordeaux on GRID'5000.

The trace from Bordeaux contains 74647 jobs, CTC has 42873 jobs and SDSC contains 15615 jobs. Thus, there is a total of 133135 jobs. In the remainder of the chapter, we refer at

the different scenarios by the name of the month of the trace for the jobs from GRID'5000, and we refer to the scenario with the jobs coming from CTC, SDSC, and GRID'5000 as "PWA-G5K".

Month/Cluster	Bordeaux	Lyon	Toulouse	Total
January	13084	583	488	14155
February	5822	2695	1123	9640
March	11673	8315	949	20937
April	33250	1330	1461	36041
May	6765	2179	1573	10517
June	4094	3540	1548	9182

Table 3.1 – Number of jobs per month and in total for each site trace.

In our simulations, we do not consider advance reservations (present in GRID'5000 traces). They are considered as simple submissions so the batch scheduler can start them when it decides to. To evaluate the heuristics, we compare simulations together so we can not compare ourselves with what happened in reality. Furthermore, note that we add a meta-scheduler to map the jobs onto clusters at submission time, as if a grid middleware is used. However, on the real platform, users submit the cluster of their choice (usually they submit to the site closest to them) so simulations diverge from reality.

The traces taken from the Parallel Workload Archive were taken in their standard original format: they also contain "bad" jobs described by [83]. We want to reproduce the execution of jobs on clusters, so we need to keep all the "bad" jobs removed in the clean version of the logs because these jobs were submitted in reality.

3.4.3 Platform Characteristics

We consider two platforms with different numbers of cores distributed among three sites. Each platform is used in an heterogeneous case (clusters differs in terms of CPU speed and number of processors). In all cases, all the batch schedulers use the same scheduling policy.

The first platform corresponds to the simulation of three clusters of GRID'5000. The three clusters are Bordeaux, Lyon, and Toulouse. Bordeaux is composed of 640 cores and is the slowest cluster. Lyon has 270 cores and is 20% faster than Bordeaux. Finally, Toulouse has 434 cores and is 40% faster than Bordeaux.

The second platform corresponds to experiments mixing the trace from Bordeaux (GRID'5000) and two traces from the Parallel Workload Archive. The three clusters are Bordeaux, CTC, and SDSC. Bordeaux has 640 cores and is the slowest cluster. CTC has 430 cores and is 20% faster than Bordeaux. Finally, SDSC has 128 cores and is 40% faster than Bordeaux.

3.4.4 Evaluation Metrics

In order to evaluate the reallocation algorithms and the behavior of the scheduling heuristics, we use different metrics. The first type of metrics is system centered metrics. The second type is user centered metrics.

– System metrics

Jobs impacted by reallocation: The percentage of jobs whose completion time is changed compared to an execution without reallocation. Only the jobs whose completion time changes are of interest for our study.

Number of reallocations relative to the total number of jobs: We give the percentage of reallocations in comparison to the number of jobs. A job can be counted several times if it migrated several times so it is theoretically possible to have more than 100% reallocations. A small value is better because it means less transfers.

– User metrics
The user metrics are obtained by comparing the results with a reference simulation. The reference simulation is an execution with identical parameters, except that reallocation is not performed.

Jobs finishing earlier: Percentage of jobs that finished earlier with reallocation than without. This percentage is taken only from the jobs whose completion time changed with reallocation. A value higher that 50% means that there were more jobs finishing early than late.

Gain on average job response time: Authors in [80] present the notion of response time. It corresponds to the duration between submission and completion of a job. Complementary to the previous one, the average job response time of the jobs impacted by reallocation relatively to the scenario without reallocation defines the average ratio that the duration of a job can issue. A ratio of 0.8 means that on average, jobs spent 20% less time in the system, thus giving the results faster to the users.

Figure 3.3 illustrates why jobs can be delayed and others can finish earlier onto a platform composed of two clusters. At time 0 a reallocation event is triggered. The task (T) is reallocated from cluster 2 to cluster 1 with a greater number of processors allocated to it according to our algorithm. Thus, some tasks of cluster 2 are advanced in the schedule. On cluster 1, as expected, the task is back-filled. However, assume that the task (F) finishing at time 6 finishes at time 2 because the walltime was wrongly defined. Thus, because of the newly inserted task, the large task on cluster 1 is delayed compared to an execution without reallocation. Note that even with FCFS reallocation can also cause delay. If a job is sent to a cluster, all the jobs submitted after may be delayed. Inversely the job that was reallocated to another cluster now leaves some free space and it may be used by other jobs to diminish their completion time.

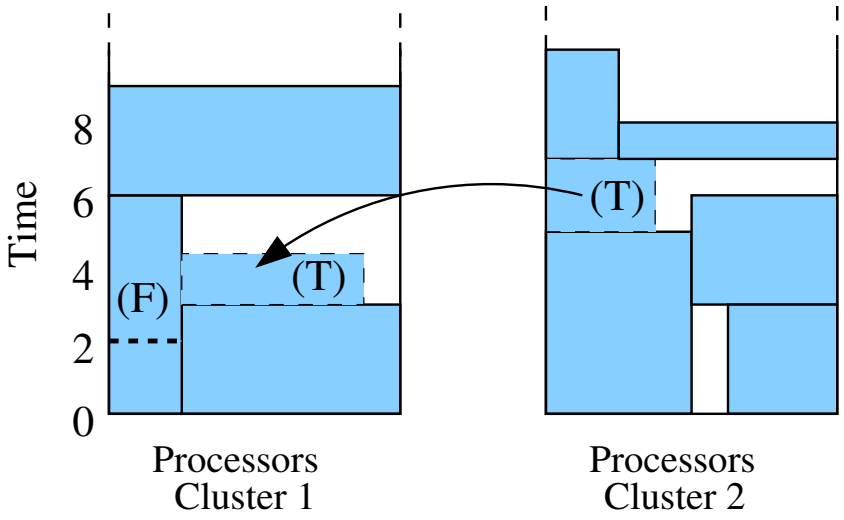


Figure 3.3 – Side effects of a reallocation.

3.5 Reallocating Rigid Tasks

In order to evaluate the different scheduling heuristics as well as the two reallocation algorithms, we ran multiple experiments using the workload traces presented in Section 3.4.2. All the jobs are submitted through the middleware, thus the middleware can reallocate any job from one site to another. The number of processors is kept as defined in the traces, only the runtime and walltime are updated in function of the speed of each cluster as defined in Section 3.4.3. Each cluster is managed by a batch scheduler using a CBF policy.

Results on heterogeneous platform where clusters are managed by a batch scheduler using a FCFS policy are presented in [45]. We also performed the experiments with homogeneous clusters (homogeneous in speed, but heterogeneous in size). Results on this can be found in the research report [49].

The experiments we ran are described in Section 3.5.1. We only present in detail the results of the experiments with heterogeneous clusters managed with a batch scheduler using a CBF policy in Section 3.5.2 because they are the most relevant to us. Indeed, CBF is a widely used batch scheduling policy and in a multi-cluster environment, clusters are usually heterogeneous. However, we give a few remarks on the FCFS platform in Section 3.5.3 and on the homogeneous platform in Section 3.5.4.

3.5.1 Experimentation Protocol with Rigid Tasks

In order to compare the different scheduling heuristics as well as the two reallocation mechanisms, we simulated the execution of the reallocation mechanism. We use both reallocation mechanisms with all six scheduling heuristics in different contexts. We studied all the combinations of algorithms with different traces of jobs in an homogeneous and heterogeneous context, as well as in a context where the clusters are managed with an FCFS policy or a CBF policy. Thus, each experiment is a tuple (reallocation algorithm,

heuristic, platform-trace, platform heterogeneity, batch scheduler). With the experiments without reallocations, this makes a total of 2*6*7*2*2+14 = 350 experiments. In this Thesis, we only give the plots on heterogeneous platforms managed by batch schedulers using a CBF policy, thus the plots correspond to 91 experiments. For all the detailed results on all experiments, refer to [49].

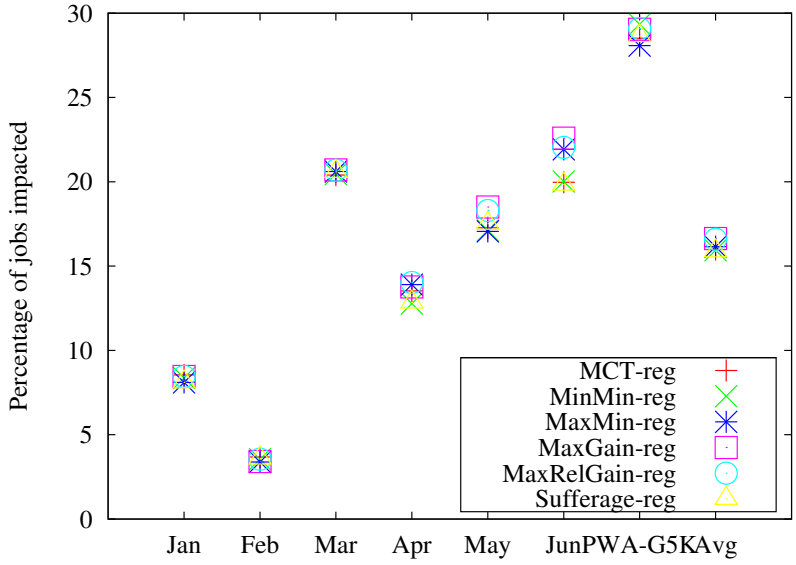
3.5.2 Results on Reallocation of Rigid Tasks on Heterogeneous Platforms

Figure 3.4(a) shows the percentage of jobs whose completion time changed because of reallocations when using the regular algorithm. Figure 3.4(b) shows this percentage when reallocation with all-cancellation is performed. Both algorithms produce similar results. The percentage depends on the trace used and is usually between 10% and 30% with an average close to 17%. The different heuristics also produce close results. This means that the impacted jobs depend on the submissions frequency. If the platform is quite empty, submitted jobs will start execution as soon as they are submitted so reallocation will not take them into account. On the other hand, when the platform is very loaded, most of the jobs do not have the opportunity to be reallocated because other waiting queues may be very loaded too, due to the MCT heuristic applied at submission time.

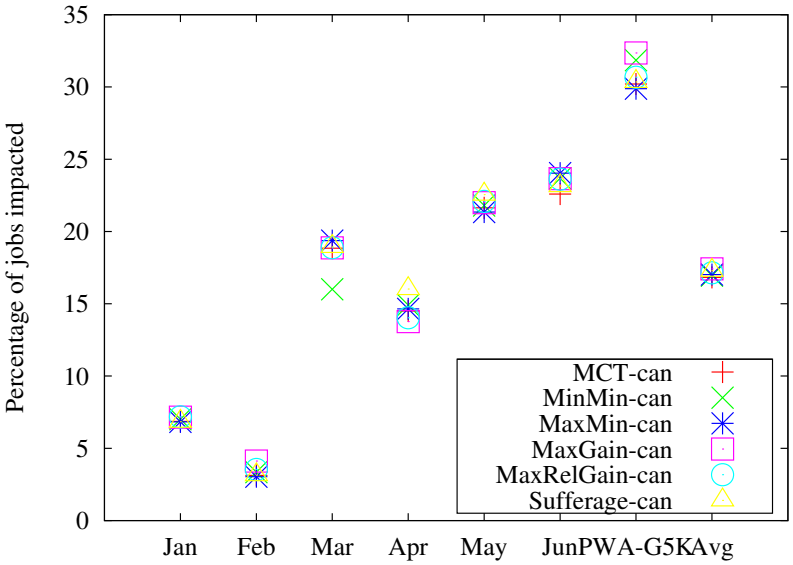
Figure 3.5(a) shows the relative number of reallocations per experiment using regular. The same job can be reallocated several times. In all cases, the number of reallocations is small compared to the number of tasks of each experiment. The small number of reallocations implies that there should not be too many migrations between clusters thus not overloading the network and local schedulers. Reallocations with all-cancellation are presented in Figure 3.5(b). There are still few reallocations compared to the number of jobs ,However, the number of reallocations is higher. With all-cancellation, all waiting queues are reduced to the currently running jobs only and when resubmitting, most jobs can migrate. Without all-cancellation, the waiting queues are longer so jobs have less chances of being reallocated. With both reallocation algorithms, the trace from February produces almost no reallocation because the MCT policy applied to all incoming jobs is sufficient to keep the waiting queues empty.

Figure 3.6(a) shows the percentage of jobs finishing earlier with reallocation than without using the regular algorithm. This percentage only takes into account impacted jobs (cf. Figure 3.4(a)). Thus, a value higher than 50 means that there is more than half of the jobs finishing earlier with reallocation than without. Concerning the heuristics, MinMin-reg gives the best results on average: all the other algorithms are around 3% behind. Figure 3.6(b) shows the percentage of jobs finishing earlier with reallocation than without using all-cancellation. Most of the time, jobs finish earlier than later, with an average of 10% gain. MCT-reg is the heuristic that produces the less jobs early on average. The other heuristics give results close to one another. On average, all-cancellation improves the results compared to regular. If we consider each experiment separately, we can see a clear improvement in the number of cases where reallocation brings a gain on the number of jobs completed earlier.

The relative average response time is compared with no reallocation on jobs impacted by reallocation. A value of 0.85 means that reallocations provide a gain of 15% on the



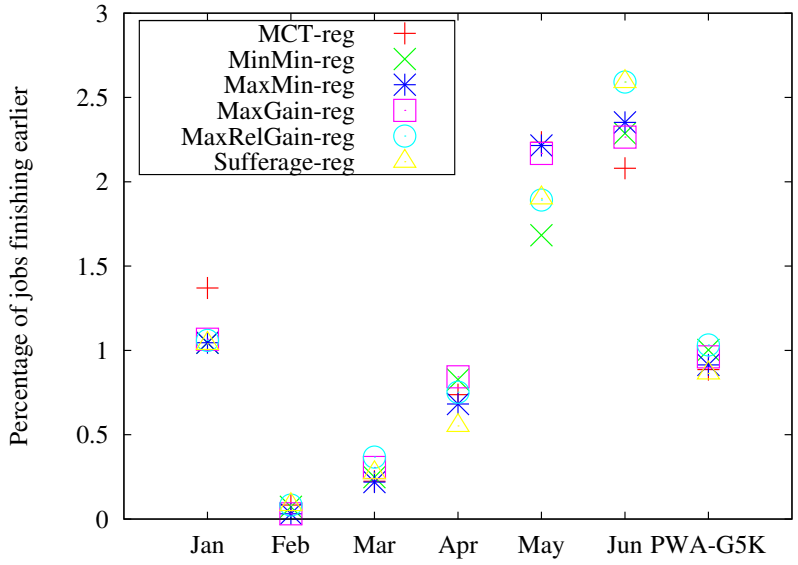
(a) regular



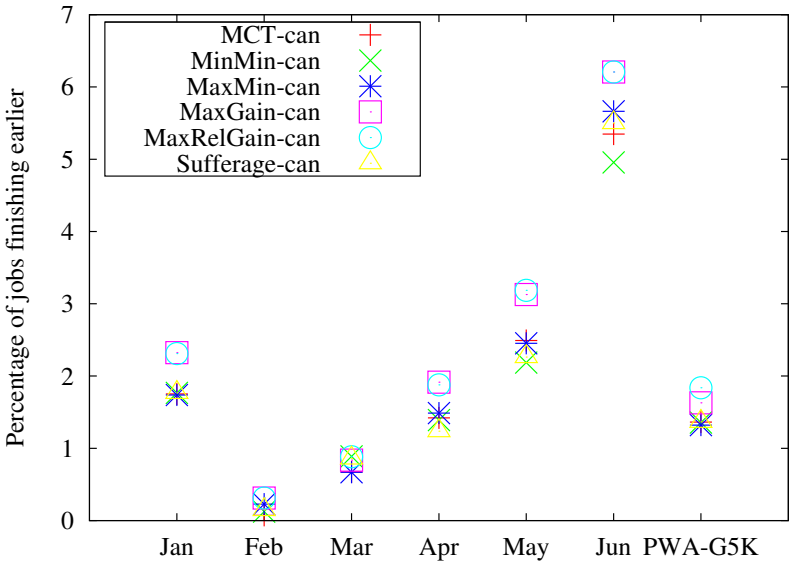
(b) all-cancellation

Figure 3.4 – Rigid jobs impacted by reallocation.

average response time of the jobs. Figure 3.7(a) shows MCT-reg has the best average. This good result comes from one experiment (January) where it improves a lot the average response time while other heuristics give less satisfactory results. Without this experiment, MCT-reg would give results close to the other heuristics on average. All heuristics are close and none has a big advantage on the others. Still, MCT-reg has a 3% advantage on the second best and can improve the relative average response time by 12% on average.



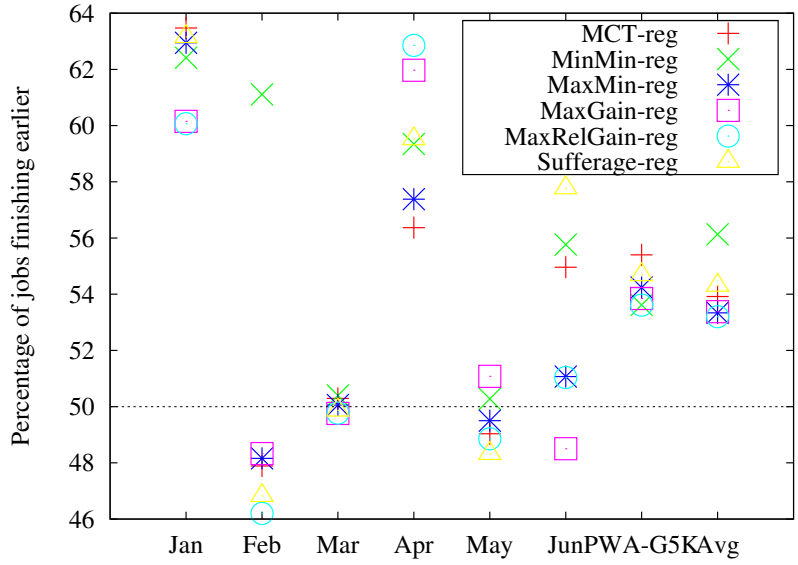
(a) regular



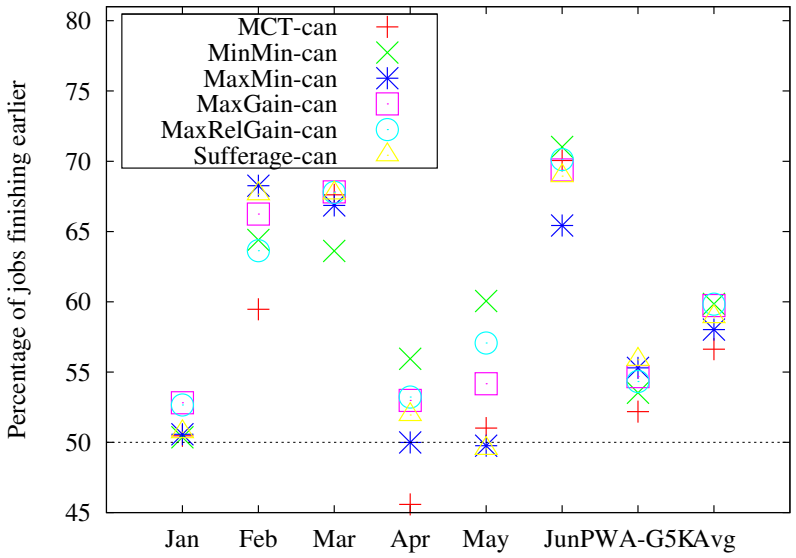
(b) all-cancellation

Figure 3.5 – Relative number of reallocations.

Figure 3.7(b) shows the results with all-cancellation. The relative average response time never increased in all experiments. All heuristics give similar results on average, with a difference between the best and the worst of less than 2%. The gain is always at least 4%, and 15% in average. Therefore, we can conclude that reallocation is a good way to take into account errors made on scheduling when tasks are submitted, even with the MCT policy applied to all incoming tasks.



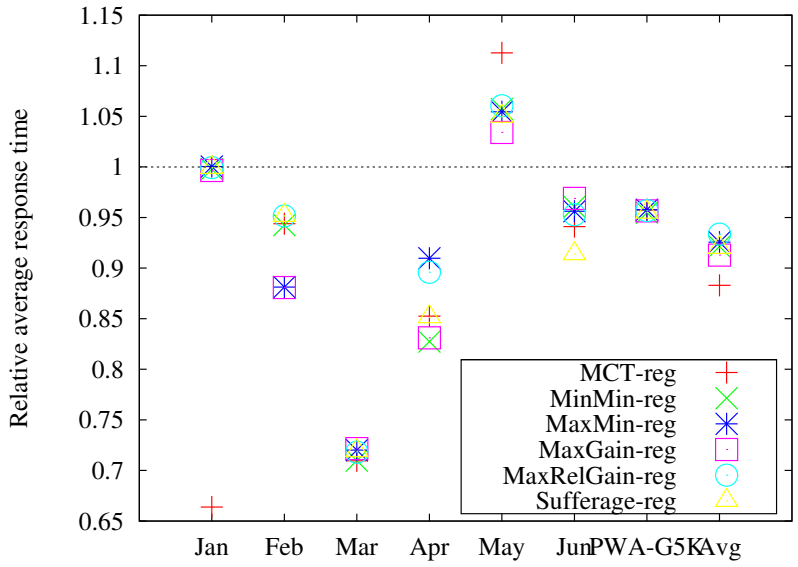
(a) regular



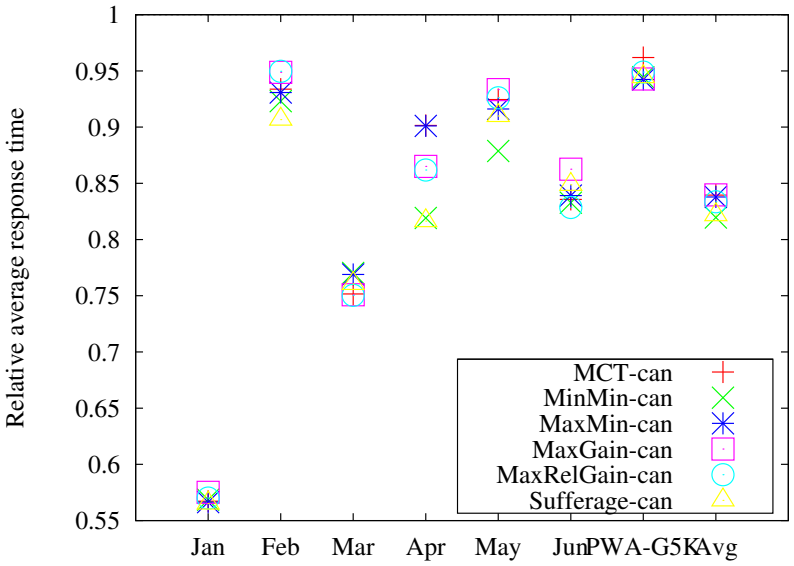
(b) all-cancellation

Figure 3.6 – Jobs finishing earlier with reallocation.

The previous figure (3.7) shows that to decrease the average response time of jobs, the choice of the scheduling heuristic can not really be made based on the results since they are too close. So, the scheduling heuristic of choice would be MCT-reg because of its simplicity of implementation compared to the others as well as its better execution time. However, concerning the reallocation mechanism, all-cancellation is a lot better. On average, it is better than regular and improves the results between 4% and 10%.



(a) regular



(b) all-cancellation

Figure 3.7 – Relative average response time.

We can conclude from the different cases studied here that all-cancellation usually brings improvement over the regular reallocation algorithm. The drawback of canceling jobs and resubmitting them is that there are more reallocations, thus batch systems are issuing more requests (submissions as well as cancellations). The other drawback of the all-canceling version of the algorithm is that it can produce starvation as presented in Section 3.3.2.1.

3.5.3 Reallocation over FCFS

FCFS is a simple scheduling algorithm that schedules jobs at the end of the waiting queue even if there is a place, earlier in the queue, large enough for the job. So, FCFS tends to have longer waiting queues than CBF. Because of that, there are more jobs to consider for reallocation, and thus, more reallocations are performed. FCFS usually produces twice as much reallocations than CBF. But, even if the number of reallocations is twice bigger, this stays small compared to the total number of jobs. The maximum number of reallocations corresponds to 13.5% of the number of jobs.

Because more reallocations are performed, more jobs are impacted by the process of reallocation. The difference with the number of jobs impacted is smaller than the difference in the number of reallocations which suggests that jobs are moved several times. While using FCFS, more jobs are impacted than with CBF. The difference is higher on homogeneous platforms than on heterogeneous platforms.

Concerning the percentage of jobs finishing earlier when reallocation is performed, it is quite close to the experiments with CBF. The difference is usually less than 3%. However, on the homogeneous platform when using all-cancellation, MinMin, MaxGain, and MaxRelGain provide good results by increasing the percentage of jobs earlier by almost 10% over CBF.

With the regular reallocation algorithm, the gain on the average response time of impacted jobs is close to the one with CBF. The gain is usually a little better with CBF (around 1% on average). However, with all-cancellation, the platform with FCFS benefits more from reallocations. Indeed, the gain is increased by almost 10%. Because CBF executes jobs faster, the gain provided by reallocation is smaller.

On a Grid where clusters are managed by batch schedulers using a FCFS policy, it is also beneficial to perform reallocations. The number of reallocations is reasonable, and the gain is not negligible. In this context, the all-cancellation mechanism has an advantage over the regular one: because all jobs are canceled, better scheduling decisions may be taken thus leaving less holes in the waiting queues. CBF does not benefit from this, because the algorithm itself fills the holes in the queue.

3.5.4 Reallocation on Homogeneous Platforms

Some of the differences between the results on homogeneous and heterogeneous platforms come from the way we implemented the heterogeneity. On homogeneous platforms, we kept the jobs duration that were in the trace files. On the heterogeneous platform, we decrease the execution time of the jobs on the faster clusters. Thus, the global throughput of the platform increases and the waiting queues are emptied faster. Therefore, on homogeneous platforms, there are more jobs waiting. However, if we decided to slow the execution of the jobs on the slower clusters, the heterogeneous platform would have had a smaller global throughput and longer waiting queues. This could change some of the following remarks on the results.

On homogeneous platforms, there are more reallocations as well as more jobs impacted, which is because, as mentioned before, the homogeneous platform global through-

put is smaller, so there are more jobs in the waiting queues to be reallocated, thus more possibilities of reallocations. Also, on the homogeneous platform, jobs can move freely between clusters, but on the heterogeneous platform, going to a slower cluster would mean an increase in execution time, so it is less probable to have a job migrating from a fast cluster to a slow one unless a lot of resources became idle on the slowest clusters thus allowing a big improvement on the start time of the application.

Regarding the percentage of jobs finishing earlier with reallocations, the homogeneous platform has better results than the heterogeneous one. With the regular algorithm, the percentage of jobs earlier is higher by around 10% in a CBF setup. With FCFS and the regular algorithm, it is also higher, but just by 2%. With the all-cancellation algorithm, the numbers are still higher on the homogeneous platform, but the previous values are reversed between FCFS and CBF: FCFS on the homogeneous platform is better by 10% and CBF is better by 2%.

The gains on the average job response time is higher on the heterogeneous platform by a few percents when reallocation is performed using the regular algorithm. However, with the all-cancellation algorithm, the gains are higher on the homogeneous platform. On an homogeneous platform, the average response time can be reduced greatly. On average, it is reduced by 25%, and in the best experiment in our simulations, it is divided by a factor of four, which is the best result obtained in all the experiments we did.

These results show that if errors of scheduling only come from the errors of runtime predictions made on the jobs it is beneficial to reallocate. On the heterogeneous platform, the prediction error changes between the clusters depending on their speed. Even if the MCT policy applied at jobs submission looks sufficient, reallocation is still very beneficial in an homogeneous environment and should be implemented in a middleware.

3.6 Reallocating Moldable Tasks

Grid services often consist of remote sequential or rigid parallel application executions. However, moldable parallel applications, linear algebra solvers for example, are of great interest but requires dynamic tuning which has mostly to be done dynamically if performance are needed. Thus, their execution on the Grid depends on a remote and transparent submission to a possibly different batch scheduler on each site, and means an automatic tuning of the job according to the local load.

In this section, our goal is to extend previous simulation results and study in details the behavior of some heuristics on moldable jobs. The previous section on reallocation of parallel rigid tasks showed that the best heuristics were MCT and MinMin. The other heuristics present in Section 3.5 gave mitigate results and have the same complexity as MinMin, we argue that they would also give poor results. Thus, we only keep MCT and MinMin.

First, we start by explaining how we manage moldable jobs in our system in Section 3.6.1, and then, we present the characteristics of the experiments in Section 3.6.2. Then, Section 3.6.3 contains the results of simulations of reallocations in heterogeneous Grids. Finally, we give some remarks on the results of simulations in homogeneous envi-

ronments in Section 3.6.4.

3.6.1 Moldable Jobs Management

In order to execute moldable jobs, it is necessary for the server component to be able to choose the number of processors of the job. To do so, we present how we implemented the research for the most suitable number of processors. Once the server is able to choose the number of processors, we present how we obtained the moldable jobs from traces containing only rigid jobs.

3.6.1.1 Moldable Jobs Management on a Cluster

In [67] authors use moldable jobs to improve the performance in batch systems. The user provides the scheduler *SA* with a set of possible requests that can be used to schedule a job. Such a request is represented by a number of processors and a walltime. SA chooses the request providing the earliest finish time. In our work, we use the same technique, but the server is able to choose itself the number of processors and walltime.

The choice of the number of processors and walltime is done by the server each time a request arrives, either for the submission of a job or for an estimation of completion time. To determine the number of processors to allocate to the job, the server performs several estimations with different number of processors and returns the best size: the one giving the earliest completion time. To estimate the completion time, the server can directly query the batch scheduler (but this capability is generally not present) or have it's own mechanism to compute the estimated completion time by simulating the batch algorithm for example.

The simplest idea to obtain the best size for the job is to perform an *exhaustive search*: For all possible number of processors (from one to the number of processors of the cluster), the estimation method provides a completion time with regard to the current load of the cluster. This method is simple and will choose the best size for jobs, however, it is time consuming. Indeed, each estimation is not instantaneous. Thus, for a large cluster, the estimation must be done a lot of times and the finding of the number of processors can require a long time.

In [159] the authors benchmark different sizes of the LU application from the NAS parallel benchmarks [7]. Their study shows a strictly increasing speedup up to 32 processors (adding processors always decreases execution time). But after this point, the execution time increases. It is due to the computation to communication ratio of the job becoming too small. This kind of job is not uncommon, thus we consider moldable jobs with *strictly increasing speedups until a known number of processors*. This maximum limit is usually obtained by benchmarking the applications with different number of processors.

Thus, in order to improve the speed in choosing the number of processors of a task, we can restrict the estimation from one processor to the limit of processors of the job. For jobs that don't scale very well, this will greatly reduce the number of calls to the estimation method thus reducing the time needed to find the most suitable number of processors.

Because of the hypothesis that speedup is strictly increasing until a maximum number of processors, we propose to perform a *binary search* on the number of processors to find

how many of them to allocate to the job. Instead of estimating the completion time for each possible number of processors, we start by estimating the time for 1 processor and for the maximum number of processors. Then, we perform a classical binary search on the number of processors. This reduces the number of estimations from n to $log_2 n$.

In particular cases the binary search will not provide the optimal result because of the back-filling. Let us consider an example in order to illustrate this behavior. Consider a cluster of 5 processors and a job needing 7 minutes to be executed on a single processor. With a perfect parallelism, this jobs needs 3.5 minutes to run on 2 processors, 2.33 on 3, 1.75 on 4 and 1.4 on 5. Upon submission, the cluster has the load represented by hatched rectangles in Figure 3.8. First, the binary search evaluates the completion time for the job on 1 and 5 processors (top of the figure) and obtains completion times of 7 and 7.4 minutes respectively. Then, the number of processors is set to 3 (middle of 1 and 5). The evaluation returns a completion time of 7.33 (bottom left of the figure). The most promising completion time was obtained with 1 processor, thus the binary search looks between 1 and 3. Finally, the best completion for the tested values time is obtained for 2 processors: 6.5 minutes (bottom right). However, the best possible completion time the job could have is 1.75 minutes with 4 processors. Indeed, with 4 processors, the jobs can start as soon as submitted, but this value was disregarded by the binary search. During our tests to verify the behavior of the binary search on thousands jobs, the results were the same as the exhaustive search which means that the "bad" cases are rare.

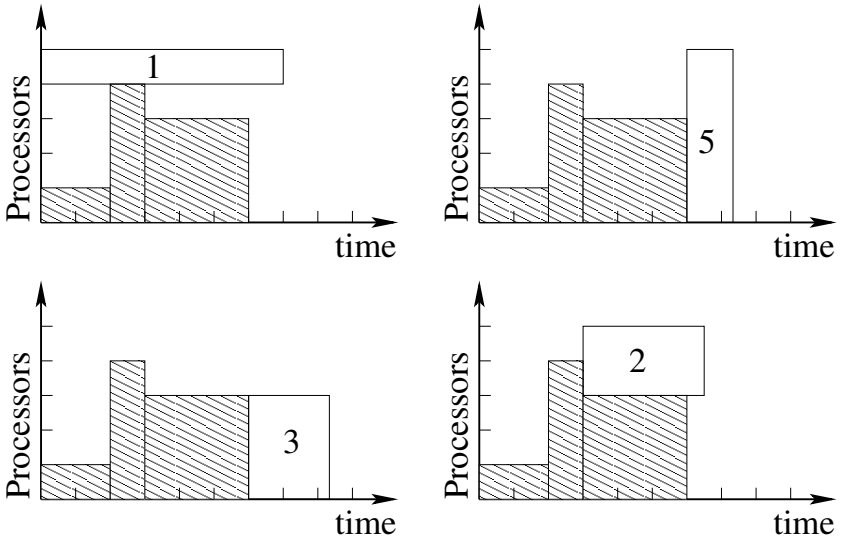


Figure 3.8 – Estimations made by the binary search.

If the maximum number of processors of a job is large, using the binary search reduces enormously the number of estimations to do, potentially by orders of magnitude. For example, if a job can be executed on 650 processors the exhaustive search performs 650 estimations of completion time and the binary search performs only 10. The binary search in this case is thus 65 times faster.

3.6.1.2 Obtaining Moldable Jobs from Traces

The jobs contained in the trace files are parallel rigid jobs. So, in order to simulate the moldable jobs, we defined 4 types of jobs using Amdahl's law ($speedup = \frac{1}{(1-P)+\frac{P}{N}}$ with P the fraction of parallel code and N the number of processors). The law states that the expected speedup of an application is strictly increasing but the increase rate diminishes. The execution time of an application tends to the execution time of the sequential portion of the application when adding more processors.

To obtain the 4 types of moldable jobs, we vary the portion of the jobs that is sequential as well as the limit of processors until the execution time decrease. The different values for the parallel portion of code are 0.8, 0.9, 0.99 and 0.999. Figure 3.9 plots the speedups for the different values of parallel code for different number of processors. Note that the y-axis is log-scaled. The figure shows that there is some point where the speedup increase becomes negligible. For the limits, we chose to use 32, 96, 256, and 650 processors. These values were chosen in accordance to the gain on the execution time of adding one processor. When the gain becomes really small, chances are that the internal communications of the job will take most of the time and slow down the task. Furthermore, the 650 limit is given by the size of the largest cluster of our simulations. So, the 4 types of jobs we consider are 4 couples (parallel portion, limit of processors): $t1$:(0.8, 32), $t2$:(0.9, 96), $t3$:(0.99, 256) and $t4$:(0.999, 650).

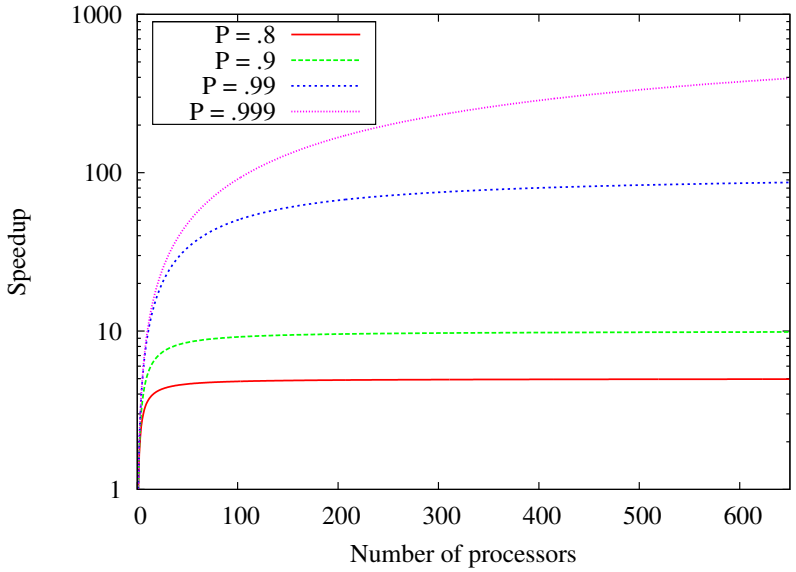


Figure 3.9 – Speedups for the Amdahl's law for different parallelism portions.

In the traces, there are more tasks using a small number of processors than tasks using a lot of processors. Thus, each job from the trace files was given a moldable type. In each simulation we present, there are 50% of jobs of type $t1$, 30% of type $t2$, 15% of type $t3$ and 5% of type $t4$. The type of a job is chosen randomly. In order to keep a more realistic set of jobs, we decided to keep the sequential jobs of the traces sequential. All the characteristics

of the jobs are summarized in Table 3.2.

Job type	Parallelism	Limit	Repartition
$t1$	0.8	32	50%
$t2$	0.9	96	30%
$t3$	0.99	256	15%
$t4$	0.999	650	5%

Table 3.2 – The four types of moldable jobs.

During the simulations, the server uses information from both the traces and the type of the job to choose a suitable number of processors and a walltime for the job. In order to do so, the server uses the binary search described in Section 3.6.1.1 to choose a number of processors and uses the following process to choose the walltime: first, it computes the speedup of the job in the trace file using Amdahl's law, the type of the job and the number of processors: $spd = amdahl(p, n_t)$ with p the parallel portion of the code and n_t the number of processors used in the trace file. Second, the server computes the walltime of the job on one processor: $w_1 = w_{n_t} * spd$. Third, the server computes the speedup of the job for the current number of processors chosen by the binary search: $spd_b = amdahl(p, n_b)$. Finally, the server computes the walltime for the job: $w_b = \frac{w_1}{spd_b}$.

To obtain the actual execution time for the moldable jobs, we keep the same difference ratio as the one in the trace file. Thus if the runtime of a job was twice smaller than walltime in the trace file, it will also be twice smaller than the walltime in the simulations, independently of the number of processors chosen for the job. Furthermore, the runtime and walltime are modified in accordance with the speed of the cluster given in Section 3.4.3.

3.6.2 Experimentation Protocol with Moldable Tasks

On real life sites, tasks can be either submitted by a Grid middleware or by local users. Thus, we investigate the differences in behavior of our mechanism depending on heuristics: on dedicated platforms, where all tasks have been submitted through our middleware; on non dedicated platforms where two third of the jobs issued from the traces are directly submitted through batch schedulers, simulating the local users.

Because we select the type of the jobs (see Section 3.6.1.2 for the different types of moldable jobs) randomly we perform all experiments 10 times with a different random seed. We also select randomly which jobs are submitted to the middleware or used as external load on the local clusters in the case of a non dedicated environment.

An experiment is a tuple (reallocation algorithm, heuristic, platform-trace, platform heterogeneity, dedicated, seed) where the seed is used to draw the type of a job in the trace, and concerning non dedicated platform, to draw if a job is submitted to the middleware or directly to the local scheduler. We used 10 different random seeds, hence, in addition to the reference experiment using MCT without reallocation, we conducted 28+2*2*2*7*2*10: 1158 experiments in total. In this thesis, we only detail the results for the heterogeneous context, thus the figures we present correspond to 574 experiments.

Executing MinMin is very slow, so to have a reasonable reallocation time, MinMin is executed only on a subset of the waiting jobs. We decided that a reallocation phase should not take more than one minute, so in this thesis, MinMin is executed on the 20 oldest waiting jobs. The estimations of completion times of all the remaining jobs have to be updated at each iteration thus limiting the jobs taken into account reduces the number of updates to perform. Because the all-cancellation algorithm needs to resubmit all jobs, it executes the heuristic on the 20 oldest jobs and then the remaining jobs are processed in their original submission order, leading to a MCT policy.

3.6.3 Results on Reallocation of Moldable Tasks on Heterogeneous Platforms

In this section, we present the results of the simulations performed to assess the performance of reallocation of moldable tasks. First, we present the results on dedicated platforms in Section 3.6.3.1. Then, Section 3.6.3.2 contains the results for non dedicated platforms.

Figures in this section show the minimum, the maximum, the median, the lower, and higher quartiles and the average of the 10 runs of each experiment. Concerning the figures in non dedicated platforms, results only take into account the jobs submitted to the Grid middleware. External jobs submitted directly to the local resource managers are not represented in the plots.

3.6.3.1 Dedicated Platforms

In this section, clusters are heterogeneous in number of processors and in speed (cf. Section 3.4.3). All requests are done to our Grid middleware, thus there is no local job submitted directly to the batch schedulers.

The percentage of jobs impacted is shown in Figure 3.10. In six experiments for the two traces March and June, extreme cases appear where almost 100% of the jobs were impacted by reallocation. This happens when the platform has a few phases with no job, during holliday pariods for example. If there are always jobs waiting, the reallocation is able to move jobs more often thus impacting a bigger portion of the jobs. Apart from these cases, the number of jobs impacted varies between the traces from 25% to 95%. All-cancellation algorithms usually impacts more jobs. MinMin-can impacts more jobs on average than the other heuristics. MCT-reg and MinMin-reg have close results.

The number of reallocations relative to the total number of jobs is plotted in Figure 3.11. All-cancellation algorithms always produce more reallocations. The regular algorithms give results inferior than 15% so the number of reallocations is quite small compared to the total number of jobs, thus limiting the transfers. However, with the all-cancellation algorithms, it is possible to go to a value as high as 50%. Because all-cancellation empties the waiting queues, more jobs have the opportunity to be reallocated. With the regular algorithms, jobs close to execution have a very small chance of being reallocated. The regular version of the reallocation algorithm is better on this metric.

Figure 3.12 plots the percentage of jobs early. In this case, 3 experiments produce more jobs late than early. In April without all-cancellation there are always more jobs late (less than 4% on average) when reallocation is performed. However in most cases, it is better to

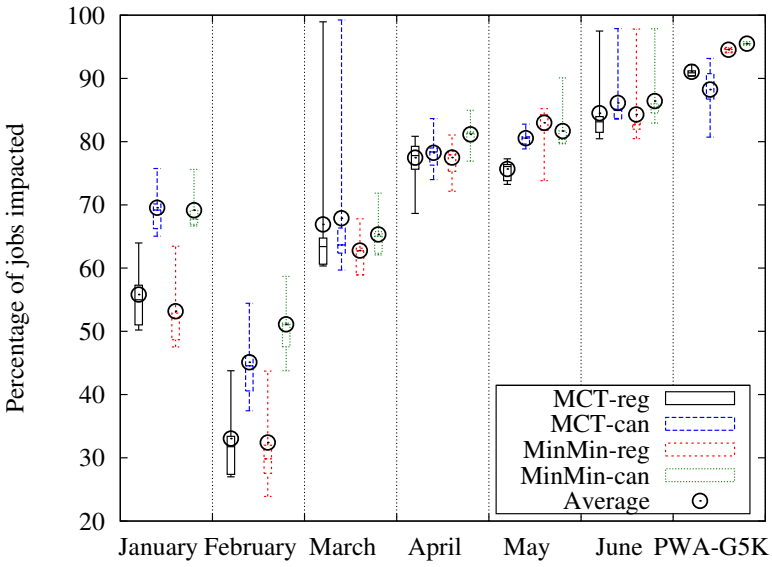


Figure 3.10 – Jobs impacted on dedicated platforms.

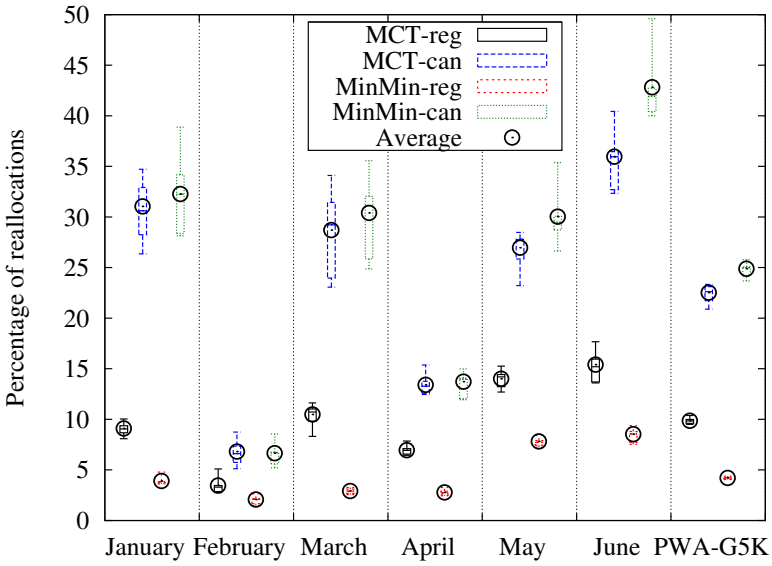


Figure 3.11 – Reallocations on dedicated platforms.

reallocate. MinMin-reg gives the worst results. It is followed by MCT-reg, then MinMin-can and finally MCT-can is the best with up to 64% of tasks early on average in March.

Concerning the average relative response time, the plot in Figure 3.13 shows a clear improvement in most cases. Excluding MinMin-reg, most gains are comprised between 10% and 40%. On average, MCT-can is the best heuristic. The reallocation without all-cancellation can worsen the average response time. It happened in 6 experiments (3 with MCT-reg and 3 with MinMin-reg). The loss is small for MCT-reg (less than 5%) thus it

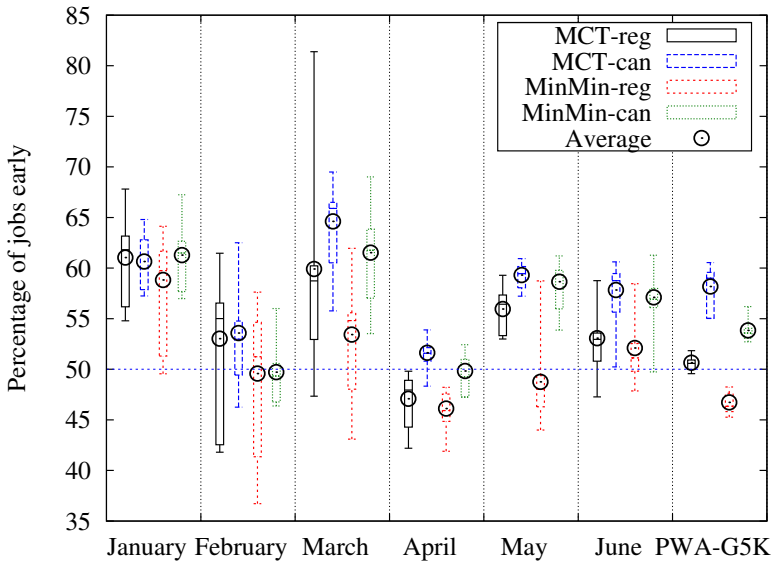


Figure 3.12 – Percentage of jobs early on dedicated platforms.

is not a problem. The all-cancellation versions are always better than their corresponding regular algorithm except in February for MCT-reg. Some experiments present a gain on the average response time while there were more jobs late than early (MCT-reg in April for example): The gains were high enough to compensate for the late jobs.

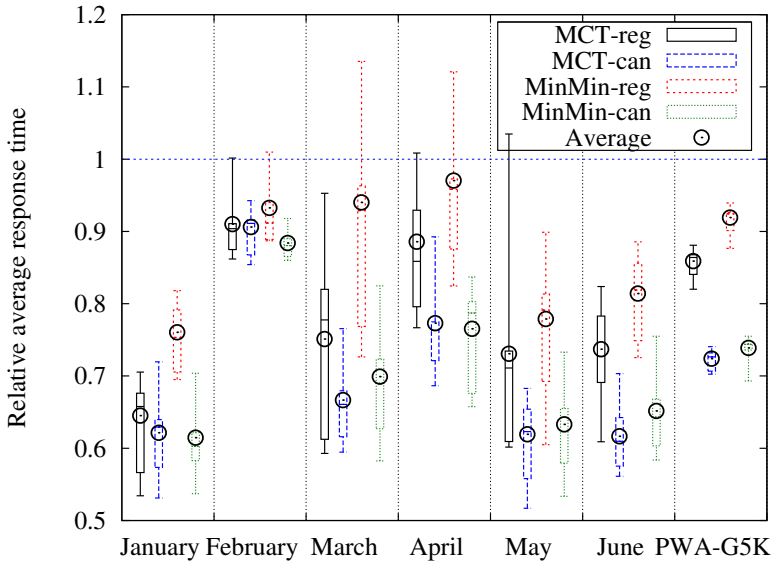


Figure 3.13 – Relative average response time on dedicated platforms.

Results in this section show that we can expect even more gains with moldable jobs that with rigid tasks presented in Section 3.5.2. Indeed, for the same platforms and jobs, gains

are more important with moldable jobs. Because moldable jobs add even more dynamism to an already uncertain environment, reallocation is even more important. When rigid jobs can not be moved because a hole in the queue is too small, a moldable one may be modified and moved, thus improving the global schedule. The MCT heuristic gives better results than MinMin because it takes more jobs into account. However even for MinMin-can where all the jobs are resubmitted, MCT-can stays better. Therefore, it confirms the result already found in Section 3.5.2 that using a simple scheduling heuristic such a MCT is enough.

3.6.3.2 Non Dedicated Platforms

In this section, we present the results on non dedicated platforms where 33% of the jobs executed on the Grid platform are moldable and submitted to the Grid middleware. Moldable jobs are chosen randomly, and the other jobs are submitted to the batch schedulers as they were defined in the trace files.

The percentage of jobs impacted by reallocation is plotted in Figure 3.14. The two all-cancellation heuristics impact more jobs than the regular ones, but the difference is really small. There is one experiment in March where MinMin-reg impacts almost all jobs: a scheduling decision taken at the beginning of the experiment impacts all the following job completion dates. For a given trace, the number of impacted jobs usually does not vary a lot.

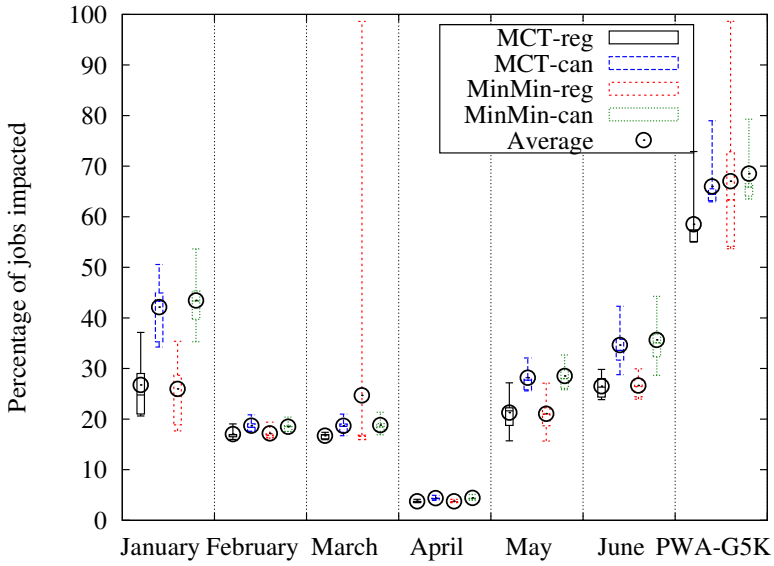


Figure 3.14 – Jobs impacted on non dedicated platforms.

Figure 3.15 plots the number of reallocations relative to the total number of moldable jobs. The number of reallocations is very small. In most cases, there are only a few dozens reallocations. The all-cancellation algorithms always reallocates more than the regular versions, but not by far. In a lot of cases, the number of reallocations corresponds to less than 1% of the number of jobs. Thus, on a non dedicated platform, the reallocation mechanism

does not produce many transfers.

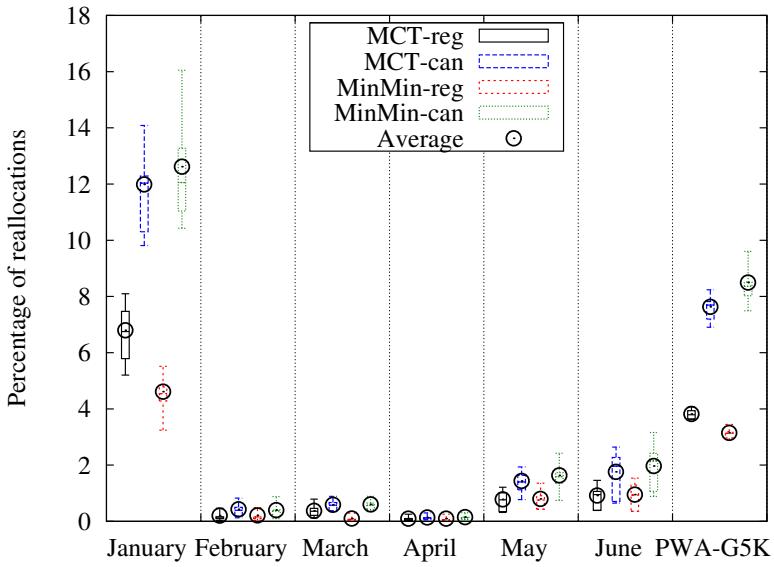


Figure 3.15 – Reallocations on non dedicated platforms.

Most experiments except the worst case for PWA-G5K and March with MinMin-reg result *in more than half of the jobs early* as we can see in Figure 3.16. The 90% jobs late in March with MinMin-reg are from the same experiment where almost all jobs were impacted in Figure 3.14. Most experiments exhibit a percentage of jobs early close to 70%. All-cancellation again produces less jobs early than regular. MCT-reg and MinMin-reg are the two heuristics of choice, but MinMin-reg gives mitigate results for PWA-G5K so MCT-reg is a better choice.

Figure 3.17 shows that the different heuristics give results close to one another on the relative average response time. All-cancellation heuristics usually have a smaller difference between the minimum and the maximum gains. Depending on the experiment, results vary a lot. In some experiments, the average response time is divided by more than two, but in other it is augmented by 40% in the worst case. However on all experiments, the average gain is positive. Thus reallocation is expected to provide a gain. Because all heuristics give very close results, the best one to use is MCT-reg. It is the simplest and the fastest to execute.

In the previous figures (Figures 3.16 and 3.17), we can see that the response time may be very different from the percentage of jobs early. In January for example, there is a majority of jobs early, and the average response time is improved. However, in June, jobs early are also dominant, but the average response time is actually worse than without reallocation. This means that the jobs that were delayed were delayed by a long time which over compensates the number of jobs early. In the PWA-G5K trace however, there are almost as many jobs early than late, but the average response time is reduced by almost 10%.

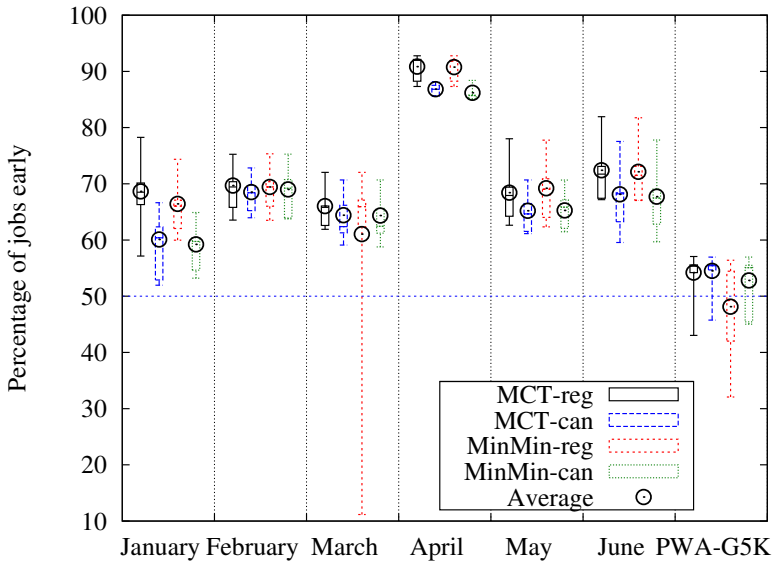


Figure 3.16 – Percentage of jobs early on non dedicated platforms.

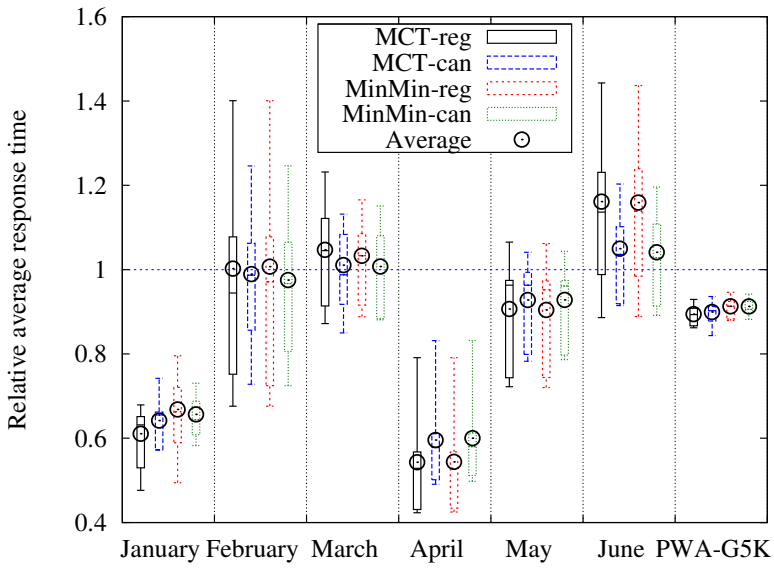


Figure 3.17 – Relative average response time on non dedicated platforms.

3.6.4 Reallocating Moldable Jobs on Homogeneous Platforms

We ran the same experiments as the ones in the previous section, but on homogeneous platforms. Clusters are homogeneous in speed but have different numbers of processors. Heterogeneity between the clusters was designed the same way as for rigid tasks in Section 3.5.4. Therefore, homogeneous platforms are also slower than the heterogeneous ones. Also, as before, the jobs can be migrated from one cluster to another more easily because

changing cluster will not change the duration of the job (except if the number of processors is changed during this process).

On a dedicated platform, the percentage of jobs impacted by reallocation is higher by often around 10% on homogeneous platform than on an heterogeneous one. In the non dedicated case, if there are more than 20% of the jobs impacted in the heterogeneous case, the homogeneous platform increases this number. However, under 20%, the results are very close on the two platforms. These observations are a logic consequence of the easier reallocation and the higher number of jobs in the queues.

More reallocations appear in the homogeneous context than on the heterogeneous one. In the dedicated environment, the relative number of reallocations is twice higher than in the heterogeneous context. The most extreme case is in June with MinMin-can where the value on this metric reaches 150% of reallocations, meaning than on average each job was moved 1.5 times. In the non dedicated environment, the number of reallocations is just higher by a few percents than on the homogeneous platform.

The percentage of jobs early is higher by few percents in the homogeneous experiments on a dedicated platform. For the non dedicated case, both platforms provide similar results. Exactly the same results emerges for the average job response time where in the dedicated case the homogeneous gives better results, and in the non dedicated environment results are very close. The same patterns as the ones on heterogeneous platform paper emerge: on dedicated platform, MCT-can and MinMin-can give the best results. MCT-reg produces less gains, and the worst is MinMin-can. On non-dedicated platform, all heuristics give similar results.

3.7 Summary

In this Chapter, we presented a reallocation mechanism used on any multi-cluster Grid without modifying the underlying infrastructure. Our solution has the same architecture than the GridRPC standard, so the implementation in a GridRPC compliant middleware is facilitated. Users requests are managed by the middleware which transforms them in submissions to a batch scheduler. The scheduling of parallel jobs can be tuned automatically by the Grid middleware each time they are submitted to the local resource manager (which implies also each time a job is migrated). We achieve this goal by only querying batch schedulers with simple submission or cancellation requests. Users ask the middleware to execute some service and the middleware manages the job automatically.

We have investigated two reallocations algorithms, the key difference between them being that one, regular, cancels a task once it is sure that the expected completion time is better on another cluster, and the other, all-cancellation, cancels all waiting jobs before testing reallocation. We also considered several scheduling heuristics to make the decision of which job to consider for migration to another site. We conducted numerous experiments and analyzed them on 4 different metrics. These experiments were conducted using a simulator compatible with the GridRPC architecture and taking as input real workload traces from different origins.

We studied two kind of jobs. First, we considered rigid jobs where the walltime of

the jobs is tuned automatically with regards to the speed of the cluster where it will be executed. Simulations results on this kind of jobs show that reallocation often provides a substantial gain on the average response time of the jobs. Indeed, in most experiments the average response time was reduced between 5% and 15% with maximum gains of a factor two. This good results are obtained without performing many reallocations: the number of reallocations corresponds only to a few percents of the number of jobs present in the middleware.

Then, we studied reallocation of moldable jobs. We started by giving a solution to manage efficiently moldable jobs in the middleware. In this case, the middleware not only adjusts the walltime of the jobs with regards to the speed of the cluster, but it also adjusts the number of processors of the jobs. With this kind of jobs, results are even better than with rigid jobs in a platform entirely managed by the middleware. The average response time is reduced by more than 20% in most experiments. In non dedicated platforms where only one third of the jobs are managed by the middleware, the diminution of the average job response time is smaller, but the gain stays non negligible nonetheless. Indeed, it can go from almost nothing to a 50% gain.

Results with both rigid and moldable jobs have similar conclusions:

First, the reallocation mechanism is profitable for users. Reallocating waiting jobs usually brings an improvement of at least 10% on the average response time of jobs. In the best cases, it can even improve this metric by 75%. The percentage of jobs early is also improved when reallocation is performed. In most setups, the users can expect more than half of their jobs to finish earlier than without reallocation. The improvement on these metrics is important for the users because they obtain their results faster. So, implementing the reallocation in a middleware will improve the user's satisfaction with the system.

The second common observation is that the all-cancellation algorithm works better than the regular one in a dedicated environment on the average response time and the percentage of jobs finishing earlier. However, all-cancellation generates more reallocations. The improvement on the average response time is usually higher by 10%, and the amount of reallocation is often doubled compared to regular, but stays small compared to the total number of jobs. Thus, in a dedicated environment, the all-cancellation should be used by the middleware in order to provide the best results to the users.

Another recurring observation is that it is not necessary to use complicated scheduling heuristics to select the order of the jobs to reallocate. The MCT heuristic that takes jobs in their original submission order provides very good results. Using offline scheduling heuristics often does not improve the results. Moreover, the reallocation time needed by such heuristic is longer and requires more network communications because it is necessary to update all the jobs information for each reallocation. MinMin provides good results in some of the experiments, but usually not enough to compensate with its long execution time. Therefore, the reallocation mechanism implemented in a middleware should use the MCT scheduling heuristic.

To facilitate the implementation, we gave directions to follow to implement the reallocation mechanism in the DIET middleware. DIET already provides most of the features required by the reallocation mechanism. The work remaining to be done is the extension of the Master Agent implementing the reallocation mechanism itself. When the it will be im-

plemented, our simulations show that DIET is expected to provide even better performance that it already is.

CHAPTER 4

Scheduling for a Climatology Application

Contents

In this chapter, we describe and evaluate different scheduling policies regarding a climatology application. In particular, we describe two layers of scheduling. A first layer is used to schedule multiple parallel simulations on a single cluster or parallel machine. The second layer is used to schedule the simulations on several heterogeneous clusters. We compare different heuristics and show that the Knapsack representation provides the best schedule on clusters while a simple repartition algorithm is really beneficial for the usage of multiple clusters.

In a later part, we describe the implementation that was realized in the DIET *middleware, and we compare the simulations with the real life experiments. These experiments were performed on the french research Grid:* GRID'5000. *Technical difficulties were encountered during experiments, so we also present the solutions we developed to solve these problems.*

4.1 Introduction

In the previous part of the thesis, we saw that providing a generic way to reallocate jobs across a Grid is beneficial to the users. In this chapter, we focus on another aspect of scheduling. We consider a Grid platform where we already have reservations done. In this dedicated environment, we want to take a particular application and propose efficient scheduling policies for its execution. We use a real life application, namely "Ocean-Atmosphere" which is a climatology application developed at CERFACS [1] (Centre Européen de Recherche et Formation Avancées en Calcul Scientifique).

The context of this study is the same as for the task reallocation presented in Chapter 3. We consider a distributed environment composed of several heterogeneous clusters. The solution we want to propose is still designed to be automatically executed by a Grid middleware. Furthermore, the application is moldable, thus the middleware decides of the number of processors allocated to Ocean-Atmosphere. However, for this study, we consider that we dispose of resources exclusively dedicated to the execution of the application. Resources may have been reserved by a resource manager and we try to optimize their usage.

The objective of this chapter is twofolds:

1. Our first concern is to provide efficient scheduling heuristics for the Ocean-Atmosphere application. The application is computation intensive and was designed to be executed on parallel machines. We want to execute several independent instances of the application differing by the input parameters. With proper scheduling heuristics, we aim at executing the application on clusters as efficiently as possible because the time allocated to an application is often limited on a parallel machine due to resource sharing constraints with other users. In order to still improve the execution time, we want to execute simulations in parallel using several clusters. Thus, we need to provide a second layer of scheduling to take into account the multiple heterogeneous clusters.
   The results of this work were presented in [43], with a detailed version available as a research report in [44].

2. The second objective regarding the Ocean-Atmosphere application is the implementation of the scheduling heuristics in a Grid middleware. Accessing a Grid can be complicated, thus providing an easy way to launch and execute the application is important. To reach this goal, we implemented the scheduling heuristics in the DIET GridRPC middleware thus providing an efficient access to the Grid to execute Ocean-Atmosphere. Also, we compare the real executions with the simulations.
   Results on this topic were published in [46]. Furthermore, a poster of this work was presented in [65]. A fully detailed version is available as a research report in [47].

1. http://www.cerfacs.fr

4.1.1 Ocean-Atmosphere

Ocean-Atmosphere is a climatology application developed at CERFACS. It uses General Circulation Models (GCM) to better understand the climate natural variability or anthropogenic effects such as climate change for the next centuries. It uses different specialized models to be as accurate as possible.

The climate simulation mainly deals with models of the atmosphere and the oceans. Ocean-Atmosphere also includes models for the continental surface through ISBA [74], sea ice with LIM [84], and river routing using TRIP [136].

The atmospheric component is the ARPEGE-Climat Atmospheric General Circulation Model [76] developed at Météo-France, the french meteorological service. The model discretizes space in order to simulate atmospheric movements. The other main component is the oceanic model used to compute heat movements in oceans. Ocean-Atmosphere uses the ORCA2 configuration from the NEMO Oceanic General Circulation Model [125] developed at CNRS, the national french research institute.

Each model discretizes space and time differently. Thus, they are coupled through the OASIS 3.0 coupler [167] which ensures time synchronization between the different General Circulation Models and performs spatial interpolations from one horizontal discretization to another. OASIS provides the coupling between the different models in the manner described in Figure 4.1: ARPEGE-ISBA sends data to TRIP to simulate the overflowing of water reserves; TRIP communicates with NEMO-LIM to simulate the discharge of rivers into seas and oceans; NEMO-LIM sends data to ARPEGE-ISBA to compute the surface temperature and the ice coverage; ARPEGE-ISBA sends the appropriate data to NEMO-LIM to model water and energy flows (except rivers).

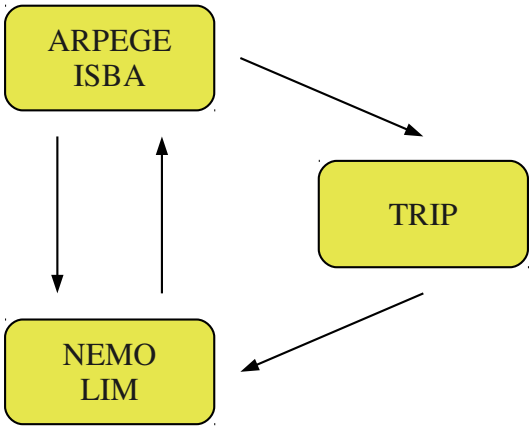


Figure 4.1 – OASIS coupling between the different models.

Imperfection of the models and global insufficiency of observations make it difficult to tune model parametrization with precision. Uncertainty on climate response to greenhouse gases can be investigated by performing an ensemble prediction. This probabilistic method consists in launching a set of simulations in parallel with varying parameters.

As it was shown in [115] with the Met Unified Model GCM, the parameter with the most notable effect on climate sensitivity is the entrainment coefficient in clouds. In this

way, each simulation has a distinct physical parametrization in clouds dynamics, with a different convective nebulosity coefficient. By comparing these independent scenarios, climatologists expect to have a better understanding of the relations between the variation in this parametrization with the variation in climate sensitivity to greenhouse gases.

Each scenario models the evolution of the present climate followed by the 21^{st} century. The present climate is simulated from 1950 to 2007 and is used to ensure that the parameters of the simulations are good enough to avoid a large deviation from reality and give better confidence in results.

An example output of Ocean-Atmosphere is given in Figure 4.2. The figure shows the (simulated) mean difference of temperature between 2100 and 2120. Simulations show a classical spatial pattern of global warming, strengthened on Northern hemisphere land points and Arctic sea.

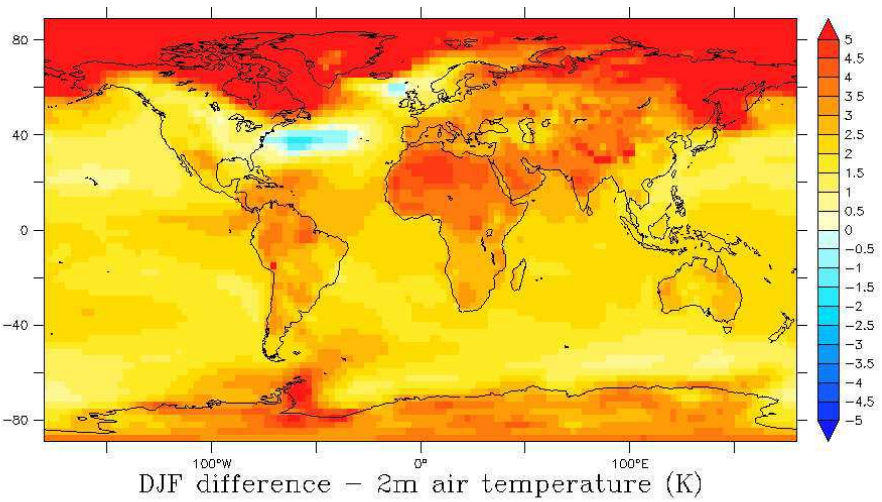


Figure 4.2 – Mean temperature difference between years 2100-2120 and control simulation.

4.1.2 Objectives Regarding Scheduling for Ocean-Atmosphere

In order execute several simulations in parallel, on one or more clusters, a large number of resources is required. The usual way to execute Ocean-Atmosphere is on a supercomputer. However, we want to perform numerous simulations in parallel, so a single machine may not suffice. Thus, using different clusters interconnected into a Grid is a solution. We use GRID'5000 as a real testbed to execute the numerous simulations. GRID'5000 has several clusters interconnected with high bandwidth networks as presented in Section 2.3.2.

In order to launch simulations easily, we developed a service in the DIET GridRPC middleware presented in Section 2.3.3.3. Furthermore, because resources usage is limited

in time, we provide efficient scheduling heuristics to have the minimum makespan.

The rest of this chapter is organized as follow: first, we present the model we derived from the application in Section 4.2. Then, we present some works on scheduling related to our model in Section 4.3. Next, we present the different scheduling strategies used to schedule the application on a cluster and on a Grid in Section 4.4. These heuristics are then evaluated in Section 4.5. Finally, we present real experiments conducted on GRID'5000 in Section 4.6 before summarizing the chapter in Section 4.7.

4.2 Architecture and Model of the Application

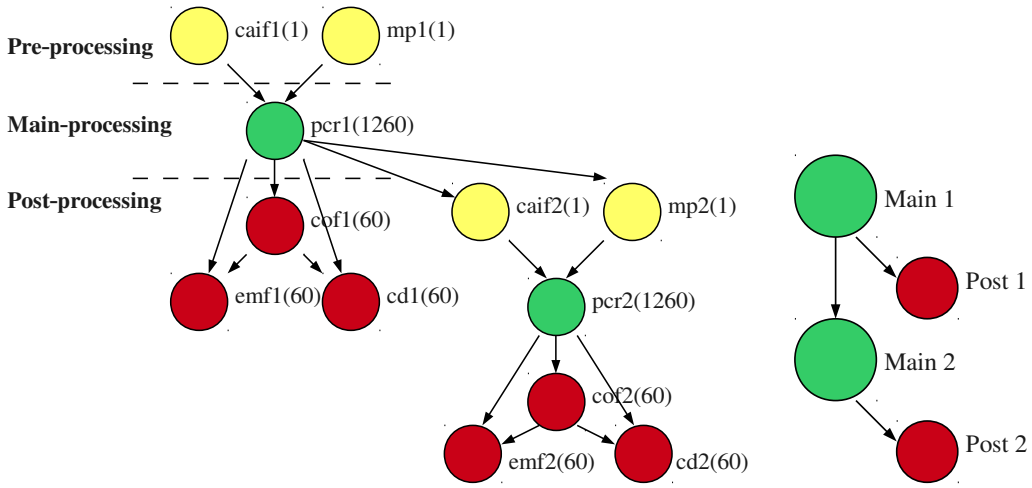
Our objective is to execute independent simulations of present climate followed by the 21^{st} century, for a total of 150 years from 1950 to 2100. Each 150 years simulation is called a scenario and takes different input parameters. Each scenario combines 1800 parallel simulations of one month each (150×12) launched one after the other. Results from the n^{th} monthly simulation are the initial conditions of the $(n+1)^{th}$ monthly simulation. An experiment corresponds to the execution of different scenarios. The number of months to be simulated and the number of scenarios may be changed by the user.

As shown in Figure 4.3(a), a monthly simulation can be divided into a **pre-processing phase**, a **main-processing** parallel task, and a **post-processing phase** of analysis. During the pre-processing phase, input files are updated and gathered in a single working directory by **concatenate_atmospheric_input_files** (caif) and the model parametrization is modified by **modify_parameters** (mp). The whole-processing phase only takes few seconds. The main-processing task **process_coupled_run** (pcr) performs a one month long integration of the climate models (ARPEGE, TRIP, NEMO, ...). The post-processing phase consists of 3 tasks. First, a conversion phase **convert_output_format** (cof) where each diagnostic file coming from the different elements of the climate model is standardized in a self-describing format. Then, an analysis phase **extract_minimum_information** (emi) where global or regional means on key regions are processed. Finally, a compression phase **compress_diags** (cd) where the volume of model diagnostic files is drastically reduced to facilitate storage and transfers.

ARPEGE code is fully parallel (using the MPI communication library), while OPA, TRIP, and the OASIS coupler are sequential applications (in the chosen configuration of our climate model). Thus, the execution time of **process_coupled_run** depends on the number of processors allocated to the atmospheric model.

Figure 4.3(a) shows the different tasks during the execution of a month simulation (nodes) and the data dependencies (edges) between two consecutive months. Numbers after the name of each task represents a possible duration of the tasks in seconds. These times have been benchmarked on a given cluster and can obviously change if the resources change (power, number of nodes, ...). However, the duration ratio between the different tasks remains the same with different hardware configurations. Given the really short duration of the pre-processing phase compared to the execution time of the main-processing task, we decided to merge them all in a single meta-task. The same decision was taken for the 3 post-processing tasks. So, in regard of the model, there are now 2 tasks: the main-

processing task and the post-processing task. Figure 4.3(b) presents the new dependencies between tasks after merging them together. As the figure shows, the problem is to schedule independent chains composed of parallel tasks.



(a) Two consecutive monthly simulations. (b) Merged tasks.

Figure 4.3 – Task dependencies (a) before; and (b) after merging tasks.

4.3 Related Work on Concurrent DAGs Scheduling

The execution of a scenario is represented by the execution of a DAG containing sequential tasks and data parallel tasks. A Directed Acyclic Graph (DAG) represents data dependencies between different tasks. Because we aim at executing several scenarios at the same time, we have both tasks and data parallelism involved in the execution of the application. Thus, our scheduling heuristics must take both aspects into account.

To schedule multiple DAGs, authors in [175] present different methods. First, it is possible to schedule DAGs one after another on the resources. Another possibility is to concurrently schedule the DAGs. It is also possible to link all the entry tasks of the DAGs to an unique entry node and do the same with the exit nodes. Then, the new resulting DAG is scheduled.

In [141], a two steps approach is described to handle the scheduling of applications presenting mixed parallelism (task and data parallelism). First, the number of processors on which a data-parallel task should be executed is computed, and then, a list scheduling heuristic is used to map the tasks onto processors. In [143], an approach of scheduling task graphs is proposed. For series compositions, the tasks are allocated the whole set of processors, while for parallel compositions, the processors are partitioned into disjoint sets on which the tasks are scheduled. In [139], a one step algorithm is proposed (Critical Path Reduction - CPR) for scheduling DAGs with data-parallel tasks onto homogeneous platforms. This algorithm allocates an increasing number of resources to the tasks on the critical path and stops once the makespan is not improved anymore. In [140], a two steps

algorithm is proposed (Critical Path and Area based Scheduling - CPA). First the number of processors allocated to each data-parallel tasks is determined. In the second step, the tasks are scheduled on resources using a list scheduling heuristic: tasks are given a rank, sorted in a list and scheduled in the order of the list. The most well known list scheduling heuristic is the Heterogeneous-Earliest-Finish-Time (HEFT) heuristic where the rank of each task corresponds to its expected completion time [165]. Authors in [60] compare 19 algorithms using makespan, average stretch, and max stretch. However, their environment is composed on several heterogeneous clusters and scheduling decisions are done using different algorithms designed for heterogeneous platforms.

On pipelined data parallel tasks, authors in [157] propose a dynamic programming solution for the problem of minimizing the latency with a throughput constraint and present a near optimal solution to the problem of maximizing the throughput with a latency constraint on homogeneous platforms. Several aspects must be kept in mind when mapping the tasks of a pipeline on the resources. Subchains of consecutive tasks in the pipeline can be clustered into modules (which could thus reduce communications and improve latency) and the resources can be splitted among the resulting modules. Resources available to a module can be divided into several groups, on which processes will alternate data sets, improving the throughput but reducing the latency.

The algorithm we present in this chapter is close to the ones presented in this section. We first compute the best grouping of processors, and then map tasks on these groups. Algorithms such as CPA and CPR are not applicable here because they would create as many groups as simulations. We show that the speedup of our application is super-linear, so, having as many groups as scenarios will be a bad solution. The algorithms would not take advantage of the repeating structure of our application. Therefore, the algorithms presented briefly in this section would lead to bad results.

4.4 Scheduling Strategies

This section present the scheduling strategies developed specifically for the Ocean-Atmosphere application. Because the scheduling is applicaiton specific, it will not be embedded in the scheduling core of the middleware, but in the client part of the application. In order to obtain execution times for the application, we start by presenting in benchmarks of the application in Section 4.4.1. Then, in Section 4.4.2 we present heuristics to schedule the application in an homogeneous context (execution on a cluster). Finally, we present an algorithm to divide the simulations across multiple clusters in Section 4.4.3 (execution on the Grid).

4.4.1 Benchmarks for Ocean-Atmosphere

The target platform is GRID'5000, so we performed the benchmarks on different clusters of this Grid. Each cluster is composed of homogeneous processors. However, clusters are different one from another. The targeted clusters either have 2 or 4 cores per node (mono-core bi-processor or bi-core bi-processor). Table 4.1 presents the hardware used on 7 clusters of GRID'5000. All clusters have a GNU/Linux operating system.

Cluster	Processor Type	# Nodes	# Cores	Memory
Capricorne	AMD Opt. 246 2.0 GHz	56	112	2 GB
Sagittaire	AMD Opt. 250 2.4 GHz	70	140	2 GB
Chicon	AMD Opt. 285 2.6 GHz	26	104	4 GB
Chti	AMD Opt. 252 2.6 GHz	20	40	4 GB
Grillon	AMD Opt. 246 2.0 GHz	47	94	2 GB
Grelon	Intel Xeon 1.6 GHz	120	480	2 GB
Azur	AMD Opt. 246 2.0 GHz	72	144	2 GB

Table 4.1 – Clusters hardware description.

Figure 4.4 shows the time needed to compute a main-task on the clusters presented in Table 4.1 depending on the number of resources assigned to its execution. The execution time of any task is assumed to include the time to access the data, the time to redistribute it to processors, the computing time, and the time needed to store the resulting data. We can see that the speedup of a main-task is superlinear: when doubling the number of resources, the time needed to execute a main-task is divided by more than two. With 5 processors on Grillon, 4205 seconds are needed and with 10 processors, only 1502. The corresponding speedup is almost 2.8. This superlinear speedup comes from the fact that a main-task needs at least 4 processors, since 3 are used by TRIP, OPA, and OASIS (one processor per model), and the remaining resources are taken by ARPEGE. Thus, with 4 processors, only one is used by ARPEGE, but with 8 processors, 5 are available for ARPEGE. This explains the superlinear speedup of a main-task. Figure 4.4 does not plot the execution time with more than 11 processors because after that point it stays constant for some time and start increasing when communications become predominant over the computations.

4.4.2 Scheduling on Homogeneous Platforms

Our goal in the context of an homogeneous platform is to minimize the overall makespan and also keep fairness among the multiple executions, meaning we want all scenarios to progress at approximately the same speed. The homogeneous platform we consider is a cluster composed of R resources and data on a cluster are available to all of its nodes.

The idea of this scheduling algorithm is to divide the R resources of the platform into disjoint groups on which main-tasks will be executed. We compute the makespan by testing each possibility for the number and the size of groups and the one leading to the smallest makespan is chosen. We assume that all main-tasks are executed on the same number of processors, meaning that all groups have the same size. Hence, all main-tasks will take the same time to be executed. Instead of giving more resources to tasks on the critical path of the application as in [139, 140], we decide of the best grouping size and then choose in which order to execute the tasks.

To obtain fairness, when a group of resources becomes idle, we schedule the next task of the less advanced DAG on this group. Since it is an homogeneous platform, scheduling the less advanced task is the same as a Round Robin. In our case, we are going to test all the possible groupings of resources, compute the makespan for each grouping and then choose

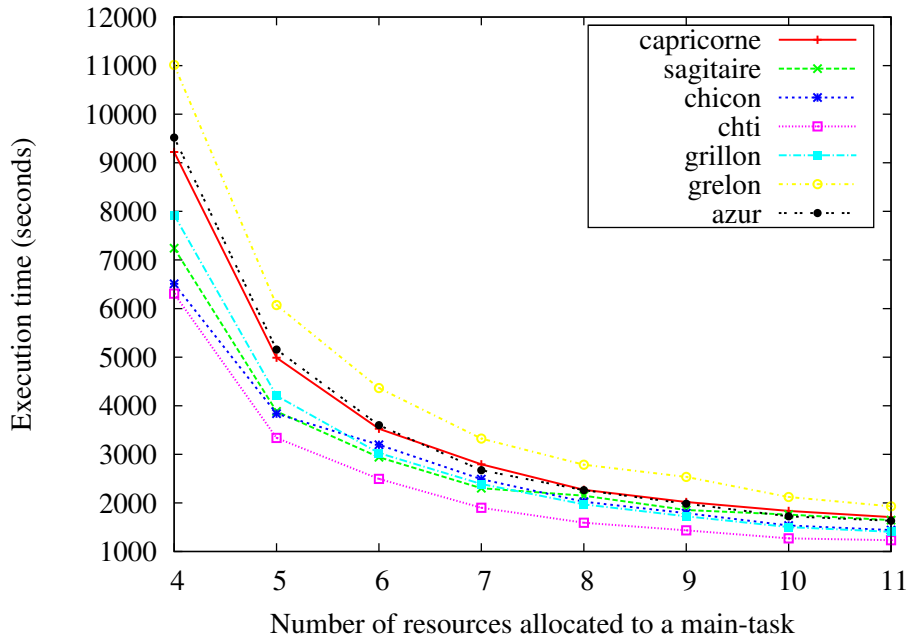


Figure 4.4 – Time needed to execute a main-task on different clusters of Grid'5000.

the best one.

4.4.2.1 Notations

To compute the makespan for a given size of group, we use the following notations:

NS - number of independent scenarios;
NM - number of months in each scenarios;
R - total number of processors;
R_1 - number of processors ($\leq R$) allocated to the main-tasks;
R_2 - number of processors ($\leq R$) allocated to the post-processing tasks;
nb_{max} - number of groups dedicated to the main-tasks execution;
G - number of processors allocated to a group to execute a main-task;
T_G - execution time of a main-task on G processors;
T_P - execution time for a post-processing task.

In order to facilitate the understanding of the equations, we define additional notations to reduce their size:

nb_{tasks} - number of each type of task ($nb_{tasks} = NS \times NM$);

nb_{last} - number of groups used on the last iteration of the main-processing tasks ($nb_{last} = nb_{tasks} \bmod nb_{max}$).

We can not process more than NS simulations simultaneously, thus we obtain the value $nb_{max} = \min\{NS, \lfloor R/G \rfloor\}$. The number of resources allocated to main-tasks is $R_1 = nb_{max} \times G$. The remaining resources are allocated to post-processing tasks, so we have $R_2 = R - R_1$.

To compute the makespan for a given group size, two cases arise depending on the number of resources. In the first case, all the processors are used for the execution of the main-tasks ($R_2 = 0$), thus post-processing tasks are postponed. In the second case, part of the post-processing tasks are scheduled to be executed in parallel of the main-tasks ($R_2 \neq 0$).

4.4.2.2 Post-processing Tasks Postponed

In this case, all the main-tasks are executed first ($R_2 = 0$), followed by the post-processing tasks. The makespan of the main-tasks is given by:

$$MS_{multi} = \left\lceil \frac{nb_{tasks}}{nb_{max}} \right\rceil \times T_G \tag{4.1}$$

If on the last iteration of the main-tasks all the groups are used ($nb_{last} = 0$), the makespan is given by:

$$MS = MS_{multi} + \left\lceil \frac{nb_{tasks}}{R} \right\rceil \times T_P \tag{4.2}$$

Figure 4.5 illustrates the schedule produced in this case.

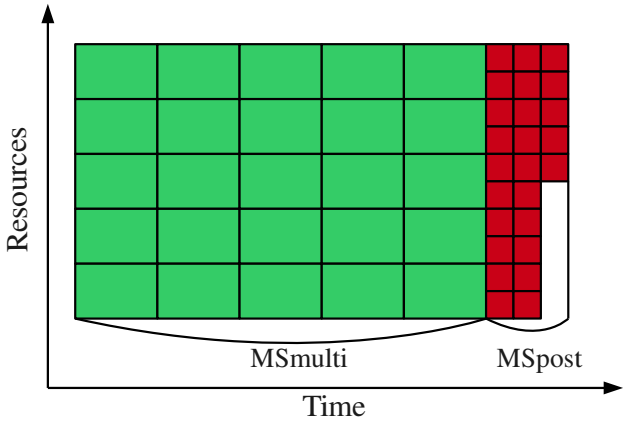


Figure 4.5 – Makespan without any processor allocated to post-processing tasks.

When the last iteration for the execution of main-tasks leaves some free processors ($nb_{last} \neq 0$), post-processing tasks start executing. A total of $remPost$ post-processing tasks do not fit on the $R_{left} = R - nb_{last} \times G$ resources left by the main-tasks. $remPost$ includes both newly produced output from the last set of main-tasks and the number of tasks that could not be executed during the last iteration:

$$remPost = nb_{last} + max\left\{0, nb_{tasks} - nb_{last} - \left\lfloor \frac{T_G}{T_P} \right\rfloor \times R_{left}\right\} \tag{4.3}$$

The makespan in this situation corresponds to the duration to process all the main-tasks and the time needed to process the $remPost$ post-processing tasks on R resources:

$$MS = MS_{multi} + \left\lceil \frac{remPost}{R} \right\rceil \times T_P \tag{4.4}$$

Figure 4.6 shows the resources left unoccupied during the last iteration of the algorithm. Post-processing tasks are scheduled on these idle resources.

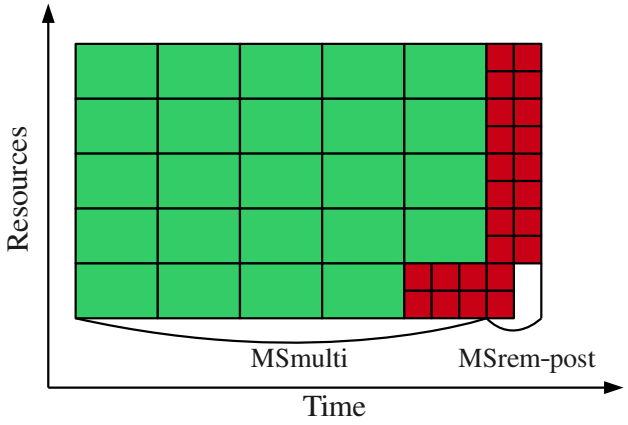


Figure 4.6 – Schedule when executing post-processing tasks with the last set of main-tasks.

4.4.2.3 Post-processing Tasks and Main Tasks Scheduled in Parallel

In this case, post-processing tasks are processed at the same time that the main-tasks ($R_2 \neq 0$). The makespan obtained by main-tasks is the same as in Equation (4.2). However, all the resources are not used so post-processing tasks can be executed at the same time.

The number of post-processing tasks that can be fully executed during the interval T_G on the R_2 resources reserved for them is defined by $N_{possible} = \lfloor T_G/T_P \rfloor \times R_2$. Comparing $N_{possible}$ and nb_{max} tells us if the R_2 resources dedicated to post-processing tasks are sufficient to execute the nb_{max} tasks produced at each iteration. If they are not sufficient, the post-processing tasks which do not fit on the resources are postponed until the end of the main-tasks' execution.

Again, two separate cases must be treated, namely the case where all resources dedicated to main-tasks are used during the last iteration ($nb_{last} = 0$) and the case where resources are not used by the main-tasks ($nb_{last} \neq 0$).

When there are no free resources during the last iteration ($nb_{last} = 0$), the number $N_{overpass}$ of post-processing tasks reported at the end is: $N_{overpass} = max\{0, (n-1) \times (nb_{max} - N_{possible})\}$. Thus, the makespan is given by:

$$MS = MS_{multi} + \left\lceil \frac{N_{overpass} + nb_{max}}{R} \right\rceil \times T_P \tag{4.5}$$

When there are idle resources during the last iteration of the main-tasks ($nb_{last} \neq 0$), a total of $N_{overpass} = max\{0, (n-2) \times (nb_{max} - N_{possible})\}$ post-processing tasks corresponding to the first $n-2$ sets of main-tasks will overpass the execution of the $n-2$

iterations already done. n is the total number of iterations needed to execute all the main-tasks. It corresponds to $n = \lceil nb_{tasks}/nb_{max} \rceil$.

Along with the nb_{max} post-processing tasks from the last complete set of simultaneous main-tasks, this gives a total of $N_{overtot} = N_{overpass} + nb_{max}$ tasks that should be scheduled starting on the resources left unoccupied in the last iteration of main-tasks ($R_{left} = R - G \times nb_{last}$). Figure 4.7 shows an example of $N_{overpass}$ and $N_{overtot}$ tasks that remain to be scheduled. The final schedule is presented in Figure 4.8.

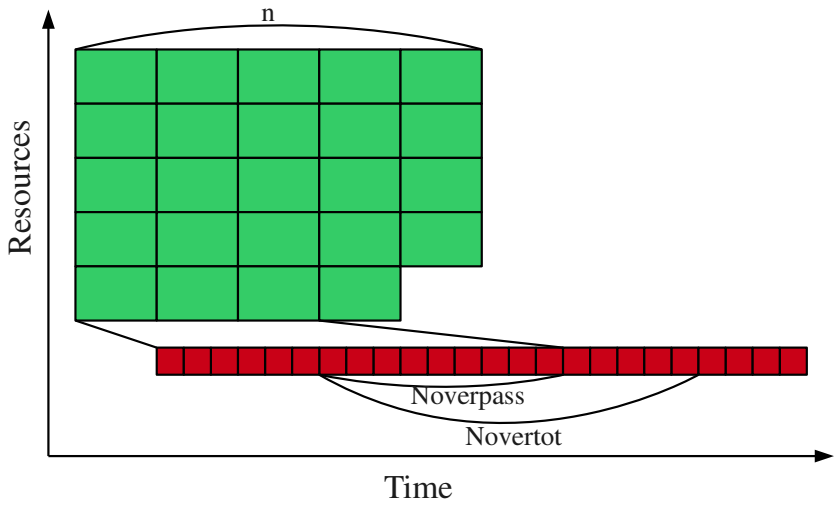


Figure 4.7 – Post-processing tasks overpassing.

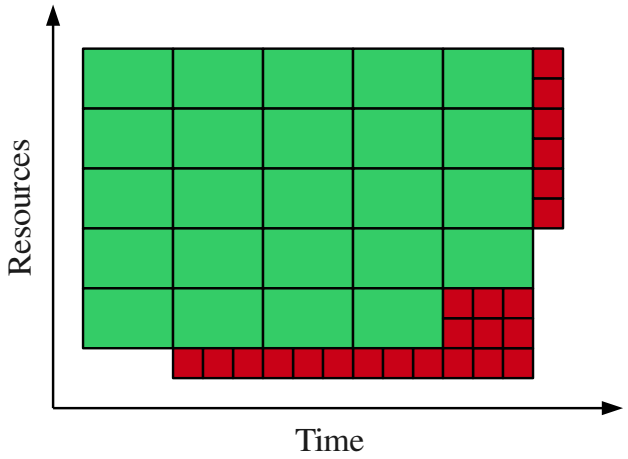


Figure 4.8 – Final schedule.

On each processor of the R_{left} remaining ones, $\lfloor T_G/T_P \rfloor$ post-processing tasks can be executed during the last iteration of main-tasks. Remaining tasks along with the newly produced post-processing tasks corresponding to the last (incomplete) set of main-tasks (nb_{last}) is:

$$remPost = nb_{last} + max\left\{0, N_{overtot} - \left(\left\lfloor \frac{T_G}{T_P} \right\rfloor \times R_{left}\right)\right\} \tag{4.6}$$

Finally, the global makespan is given by:

$$MS = MS_{multi} + \left\lceil \frac{remPost}{R} \right\rceil \times T_P \tag{4.7}$$

The 8 possibilities for the parameter G from 4 to 11 (Ocean-Atmosphere needs at least 4 processors to work properly) are tested using the previous formulas and the one yielding the smallest makespan is chosen. The optimal grouping for a number of resources going from 11 to 120 for a number of 10 scenarios is plotted in Figure 4.9. After 112 resources, the grouping stays constant. This is because there are enough resources to have 10 groups of 11 processors, plus one used for the post-processing tasks. Thus, it is not possible to use more resources.

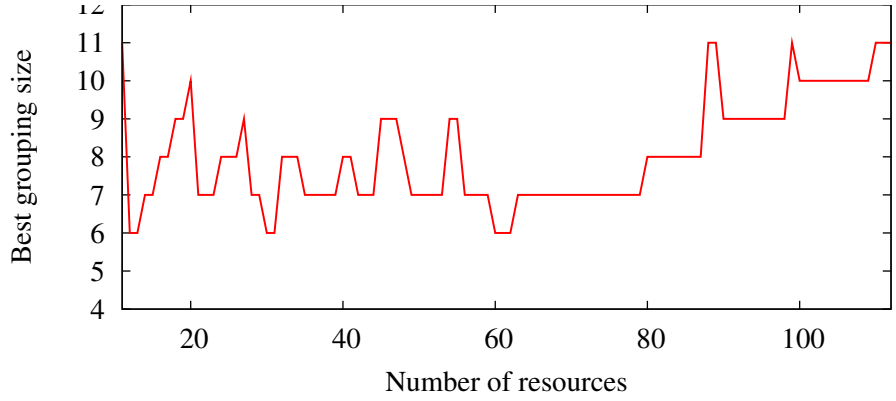


Figure 4.9 – Optimal groupings for 10 scenarios.

4.4.2.4 Improving the Schedule

For a given grouping according to the previous heuristic, it may be possible that, for a set of concurrent main-tasks and the associated post-processing tasks, all the available resources are not used. It can happen when all resources are not used for the execution of the main-tasks as it can be the case in Section 4.4.2.3. For $R = 53$ resources, and 10 scenarios, the optimal grouping is $G = 7$. Hence a total of 7 main-tasks can run concurrently, occupying 49 resources. The corresponding post-processing tasks need only 1 resource, which leaves 3 resources unoccupied during the whole computation.

We present two optimizations of the previous heuristic using the same grouping technique but which manages remaining resources differently, and a third heuristic that uses a Knapsack representation to choose the grouping.

Optimization 1. In order to improve the makespan, the unoccupied resources can be distributed among the other groups of resources. Considering the previous example, we can

redistribute the 3 resources left unoccupied among the 7 groups of resources for the main-tasks. The resulting grouping is 3 groups with 8 resources and 4 groups with 7 resources and 1 resource for the post processing tasks.

Optimization 2. Given that the post-processing tasks have a small duration and that the main-tasks have a superlinear speedup, another possibility to reduce the makespan is to use the resources reserved for post-processing tasks for main-tasks and to leave all the post-processing at the end. That way, we avoid that the resources used to compute the post-processing become idle while waiting for new tasks to process.

Optimization 3. The previous improvement looks efficient, but it is possible that the best grouping is not a regular one. In some cases, for example for 16 resources, it is better to have two groups of respectively 9 and 7 resources, instead of two groups of 8.

The optimal repartition of the R processors in groups on which the main-tasks should be executed can be viewed as an instance of the Knapsack problem with an extra constraint: no more than NS simulations can be executed simultaneously. Given a set of items with a cost and a value it is required to determine the number of each item to include in a collection such that the cost is less than some given cost and the total value is as large as possible. Using a Knapsack representation of a problem has been studied in numerous areas such as scheduling [124] and aerodynamics [149].

In our case, there are 8 possible items corresponding to groups of 4 to 11 processors. The cost of an item is represented by the number of resources of that grouping. The value of a specific grouping G is given by $1/T[G]$, which represents the fraction of a main-task being executed during a time unit for that specific group of processors. The total cost is represented by the total number of resources R. The goal of the division of the processors in groups is to compute the biggest fraction of the main-tasks during a time interval.

There are n_i unknowns (i from 4 to 11) representing the number of groups with i resources which will be taken in the final solution. Equation (4.8) has to be maximized under the constraints described in equations (4.9) and (4.10).

$$\sum_{i=4}^{11} n_i \times \frac{1}{T[i]} \tag{4.8}$$

$$\sum_{i=4}^{11} i \times n_i \leq R \tag{4.9}$$

$$\sum_{i=4}^{11} n_i \leq NS \tag{4.10}$$

Solving this linear program with few variables is very fast and takes less than a second. The n_i are integers and can only be between 0 and NS and the number of variables is small. Even with NS really greater than 10 (the number of scenarios that we want to schedule on GRID'5000 for our real experiments), the execution of the program still takes less than a second. Furthermore, the grouping given by the linear program is the optimal one for the main-tasks, except for the last iteration. The linear program only takes into consideration the main-tasks so the post-processing tasks are executed at the end.

Results of simulations comparing the basic heuristic and its optimizations on different metrics are presented in Section 4.5.1.

4.4.3 Scheduling on Heterogeneous Platforms

The second objective regarding the Ocean-Atmosphere application is to execute it on the Grid. It is thus necessary to propose a scheduling method in order to execute the scenarios on several clusters in parallel and thus take advantage of the Grid's distributed architecture. For this purpose, we use a repartition algorithm described in Algorithm 3 in order to divide the simulations into several subsets and distribute the subsets on several clusters.

Algorithm 3 input parameters are: n, the number of clusters, and "`performance`" a matrix initialized prior to the execution (or at the very beginning) by each cluster using the performance evaluation given in the previous section (Section 4.4.2). The "`performance`" matrix is filled by calling the Knapsack scheduling algorithm on each cluster for each possible number of scenarios (thus from 1 to NS for each cluster). The time needed by cluster C_i to execute x scenarios, is thus given in `performance`$[i][x]$.

Once input parameters are initialized, the algorithm can be executed. It will add scenarios one by one on the clusters: First, the number of scenarios attributed to each cluster ($nbDags$) is set to 0. Then, each scenario is scheduled on the cluster on which the total makespan increases the less, thus selecting fast clusters first. When all the scenarios are scheduled, the scheduling is returned.

Algorithm 3 DAGs repartition on several clusters.

for $i = 1$ to n **do**
  $nbDags[i] = 0$
for $dag = 1$ to NS **do**
  $MSmin = +\infty$
  $clusterMin = 1$
  for $i = 1$ to n **do**
    $temp = performance[i][nbDags[i] + 1]$
    if $temp < MSmin$ **then**
      $MSmin = temp$
      $clusterMin = i$
  $nbDags[clusterMin] = nbDags[clusterMin] + 1$
  $repartition[dag] = clusterMin$
Return $repartition$

The repartition done by this algorithm takes into account the heterogeneity between clusters. Indeed, the performance evaluation is done independently on each cluster, thus times can be adjusted accordingly using the values of the benchmarks presented in Section 4.4.1. The repartition produced is optimal with the hypothesis that once a scenario is launched on a cluster it can not be moved on another one. We do not take migrations into account because algorithm should do a lot more computations to obtain a good schedule. Its execution time would become really long if computing all the possible data transmissions.

Each cluster must compute the performance estimation NS times before executing the repartition algorithm. Because a main-task cannot be executed on more than 11 resources, it is possible to compute the number of scenarios that can be assigned on a cluster without increasing the makespan, thus reducing the number of performance estimation that must be made. If there are 40 resources, scheduling 1, 2, or 3 scenarios at the same time will not change anything. 33 resources will be used to execute the main-tasks and the overall makespan according to our modeling will stay the same.

Let C_i be a cluster currently estimating the performance, R_i the number of resources on this cluster, $T_{post}(1, C_i)$ the time needed to execute one post-processing task on one processor on the cluster and $T_{main}(11, C_i)$ the time needed to execute a main-task on 11 processor on the cluster. To compute how many scenarios can be scheduled on the cluster without degrading the makespan, $nbDags$ must be maximized under the constraints given in equations 4.11 and 4.12:

$$\underbrace{11 \times nbDags}_{\text{main}} + \underbrace{\left\lceil \frac{T_{post}(1, C_i) \times nbDags}{T_{main}(11, C_i)} \right\rceil}_{\text{post}} \leq R_i \tag{4.11}$$

$$nbDags \leq NS \tag{4.12}$$

With the knowledge of the number of scenarios that are not going to slow down the execution, it is possible to schedule this number directly on a cluster instead of one by one. For the remaining scenarios, we can use the same technique as described earlier in Algorithm 3. Doing so diminishes the number of performance estimations made by the algorithm. $nbDags$ can also be used on each cluster to know how many simulations can be executed without deteriorating the makespan. This allows to execute the evaluation heuristic just once for $1 \rightarrow nbDags$ instead of $nbDags$ times when computing the performance estimation.

4.4.4 Two-Level Scheduling

We perform a two-level scheduling: the first level considers a homogeneous platform such as a cluster where we divide resources into groups to execute the main-tasks. The second level corresponds to a multi-cluster platform where we divide the scenarios into subsets to execute them on different clusters. Thus, the local scheduling chooses a grouping of resources while the global scheduling chooses a repartition of scenarios among clusters. This is very close to the two-level architecture presented in Chapter 3.

The different steps of the execution on several clusters are displayed in Figure 4.10: (1) The client sends a request to the agent to find an appropriate server; (2) The agent looks for servers offering the wanted service; (3) Each server computes an estimation vector containing the time needed to compute from one to NS simulations using the Knapsack representation given in Section 4.4.2.4; (4) The performance vector is sent back to the agent; (5) The client receives the estimation vectors; (6) It computes the repartition of the scenarios among the clusters; (7) The client sends multiple requests to the servers; (8) Finally, each server computes the scenarios it has been assigned.

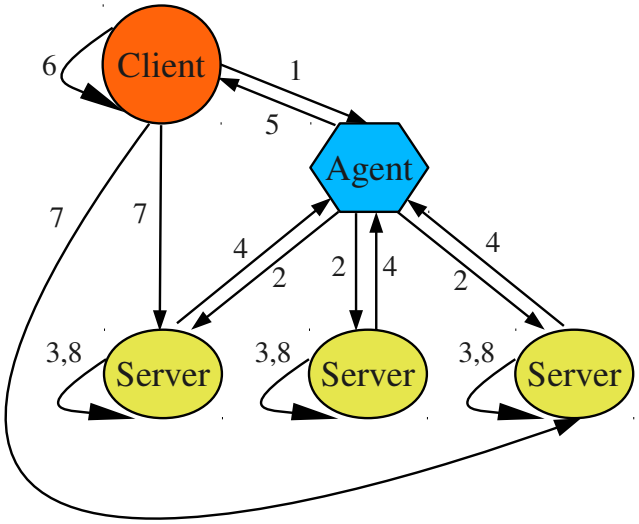


Figure 4.10 – Communications between the different entities to execute Ocean-Atmosphere.

4.5 Performance Evaluation of Scheduling Strategies

In Section 4.4 we presented different heuristics to compute the grouping of the processors in order to minimize the makespan of the scenarios on a single homogeneous cluster. To evaluate the different heuristics, we perform simulations comparing them together in Section 4.5.1. Then, in Section 4.5.2, we compare the makespan produced by the repartition algorithm compared to a Round Robin on a multi-cluster platform.

For both performance evaluations, we simulate the grouping heuristics and compute the makespan from the particular grouping. We use the execution times benchmarked on Grid'5000 presented in Section 4.4.1.

4.5.1 Grouping Heuristic vs. Grouping Heuristic

To compare the grouping heuristics, we perform several independent simulations on 5 different clusters using benchmarks of five clusters of GRID'5000: Azur, Chicon, Chti, Grillon, and Sagittaire (see Table 4.1). Thus, in the following figures, results we present are averages.

4.5.1.1 Basic Heuristic vs. Optimizations

We start by comparing the gain on the makespan obtained with the optimized heuristics with respect to the original grouping heuristic. Gains are plotted in Figure 4.11. The Figure shows the average, and standard deviation for the 5 simulations.

The representation as an instance of the Knapsack problem yields to the bests results with low resources, especially until 40 processors. In such a case, it often shows consequent gains compared to the other heuristics. In few cases, there is no gain because all

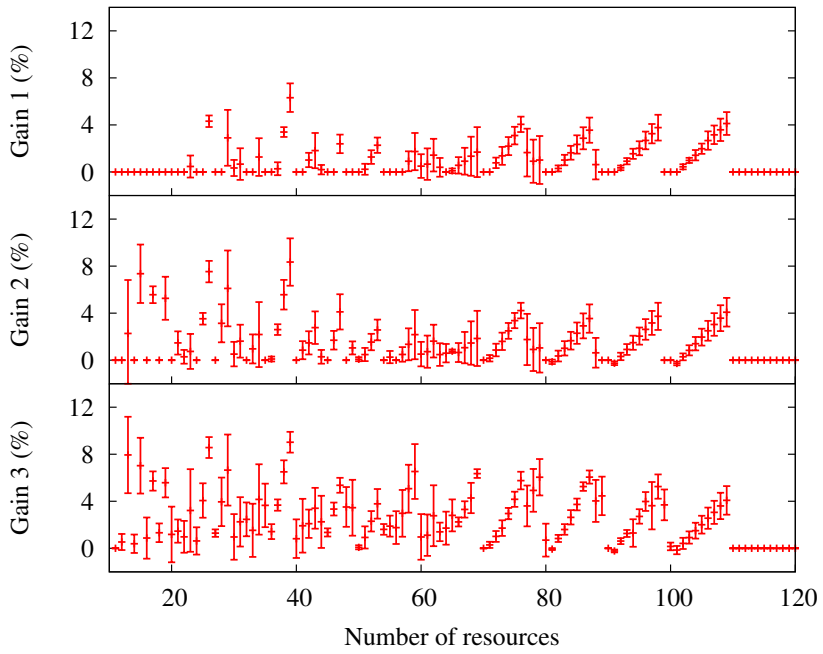


Figure 4.11 – Gains obtained by using resources left unoccupied (Gain 1), using all resources for main-tasks (Gain 2), and using the Knapsack representation (Gain 3).

heuristics produce the same grouping. When the number of resources increases, the gain decreases, but Knapsack is still the best or is a tie with another heuristic. In rare cases, it is possible to have a negative gain because of the post-processing tasks. The basic heuristic executes them at the same time as the main-tasks while the other heuristics postpone all post-processing tasks to the end. Over 112 resources, all heuristics give the same results by creating 10 groups of 11 resources and one resource to execute post-processing tasks. Another characteristic shown by this figure is that Gains 2 and 3 have a high standard deviations compared to Gain 1 for a number of resources inferior to 55. This means that these optimizations are more sensitive to the clusters speed. On average, the Knapsack representation is better than all the others when considering the execution time. Thus it is better to use the Knapsack representation to compute the grouping of processors.

We also compare the impact of the addition of new resources to the cluster on the behavior of the heuristics. Figure 4.12 plots the consequence on the execution time when adding resources. With the basic heuristic, adding resources does not always have the same consequences: when passing from 26 to 27, there is a consequent decrease of the execution time, but from 28 to 29, the execution time does not change. When redistributing the idle resources to execute the main-tasks, each addition of processor decreases the makespan, but not uniformly. Using all resources to main-tasks decreases the makespan in a more regular manner. Finally, the knapsack representation behaves almost as previously, but the curve is even smoother, meaning that each addition of resource is more taken into account by this heuristic. Again, Knapsack is the best heuristic.

Finally, we also studied the execution time of the different heuristics to find a group-

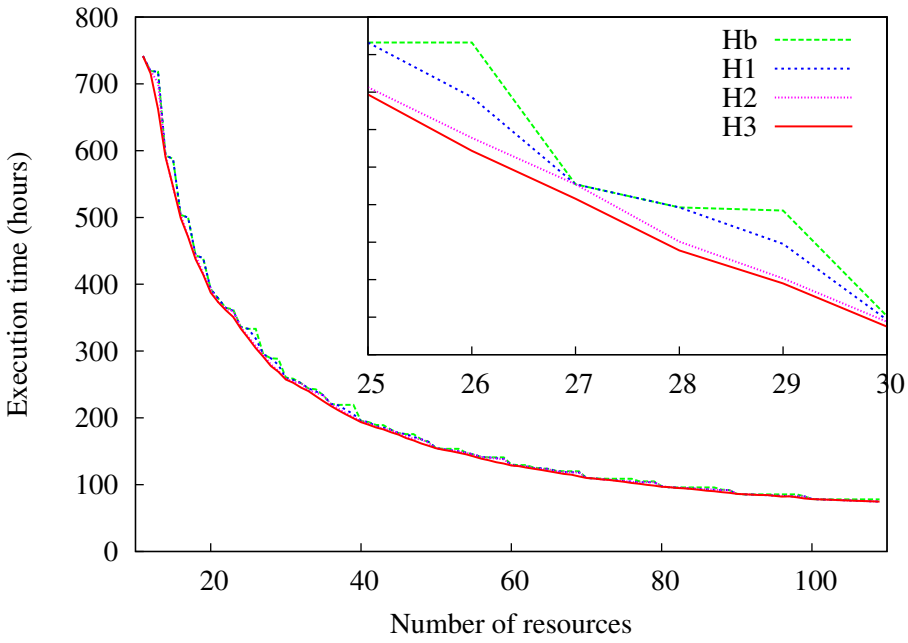


Figure 4.12 – Impact on execution time when adding resources with the basic heuristic (Hb), with redistributing idle resources (H1), with redistributing all resources (H2), and using Knapsack representation (H3).

ing. We compare the execution time of each heuristic for a number of scenario going from one to 1000 and the number of resources comprised between 11 and 1000. Time measurements were done on a laptop with a 2.2 Ghz Core 2 Duo. To solve the linear constraints of the Knapsack representation, we used `lp_solve` [2] a Mixed Integer Linear Programming (MILP) solver. The basic heuristic is always very fast to find a grouping. The slowest time took 0.0042 seconds and the fastest time was 0.0001 seconds. The two heuristics based on redistributing the resources after using the same grouping algorithm as the basic heuristic have no time difference. Solving Knapsack is slower but still fast. The slowest evaluation took 0.0997 seconds and the fastest 0.0004 seconds. So, for each grouping heuristic, the execution time to find a grouping is very small even with a very high number of scenarios. However, the computation of the makespan for a given grouping can take some time (up to a few minutes) but is not dependent on the method used to compute the grouping. Thus, the method defined in Section 4.4.3 to reduce the number of times the makespan estimation is done is useful.

4.5.1.2 Basic Heuristic vs. Knapsack in Details

The Knapsack representation yields to the best results considering the execution time, it makes better use of each resource, and has a very good execution time. To have a more detailed idea of the behavior of this heuristic, Figure 4.13 shows the different gains obtained

2. `http://lpsolve.sourceforge.net/5.5/`

in comparison with the basic heuristic. The figure shows the maximum, average, and minimum gains obtained by the heuristic. Between the maximum and the minimum gains, there are differences. It means that the gains are sensitive to the cluster performance. The diminution of gain occurs because the basic heuristic behaves better. Indeed, it makes the same grouping on each cluster since it divides resources without regarding performance. Thus, in some cases, it will make a grouping close to the one found by the Knapsack representation. We can note that the Knapsack representation produces results varying a lot depending on the cluster, but it almost never behaves worse than the basic heuristic. When it does, it is less than 0.5%, which is negligible. The large differences are in accordance with the high standard deviation presented in Figure 4.11.

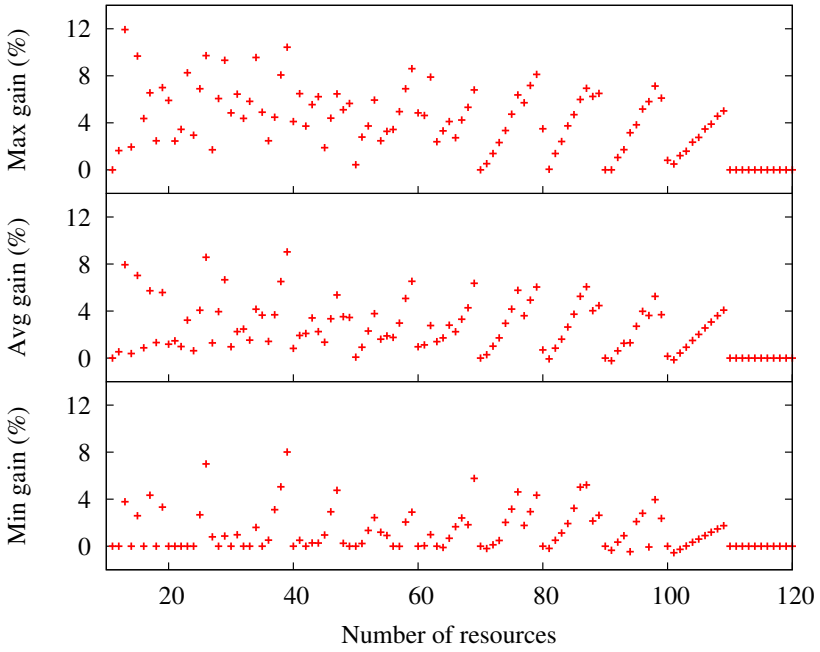


Figure 4.13 – Gains obtained with the Knapsack representation compared to the basic heuristic (Min, Average, and Max).

Now, we compare the heuristics in a distributed environment on several clusters with the repartition done using Algorithm 3, thus we compare the heuristics with the two-level scheduling presented in Section 4.4.4. Figure 4.14 shows the gains obtained by the Knapsack in this context compared to the basic heuristic. Clusters have all the same number of resources. The simulation is done with 10 scenarios. The x-axis represents the number of clusters and the number of resources per cluster. Hence, 2.25 represents the results for two clusters with 25 resources each. This simulation shows that it is possible to attain a 12% gain, but in most cases, the gains are between 0 and 8. 51% of the simulations benefit from this representation. However, no experiment is worsened by it. Using the Knapsack representation, the grouping of processors is different from the grouping with the basic heuristic. Therefore the makespan is different on each cluster, so the repartition of the scenario may be different. We can note that with several clusters, there are phases when there are no

gains at all. This happens when both heuristics produce the same grouping on the slowest cluster. For example, the slowest cluster has just one scenario, so both heuristics give the same grouping. While the overall makespan is the same with both techniques, other clusters may have a different grouping. So, even if there is no makespan improvement, the other resources usage is likely to be improved.

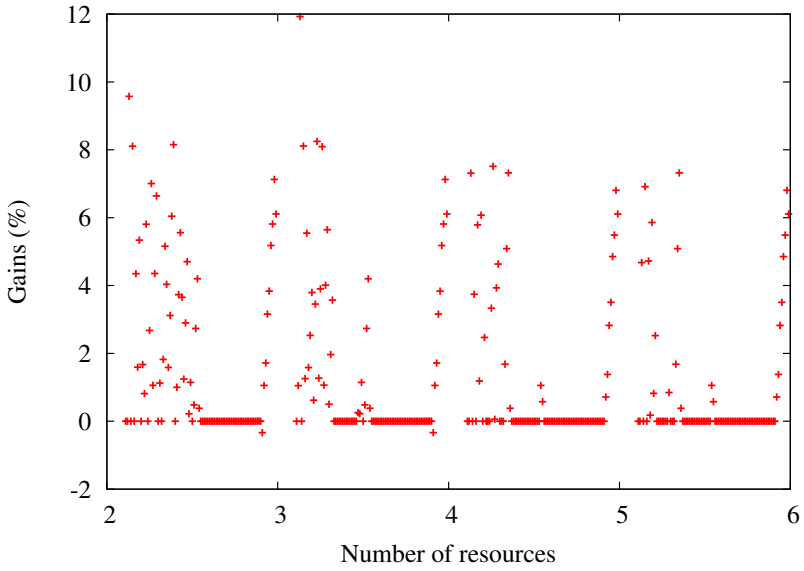


Figure 4.14 – Gains obtained by Knapsack on several clusters compared to the basic heuristic.

4.5.2 Repartition Algorithm vs. Round-Robin

To evaluate the performance of the repartition algorithm, we used the execution time of benchmarks given in Section 4.4.1 made on 5 clusters of Grid'5000. We considers that each cluster has the same number of processors. In this section, we compare the repartition done on the clusters using Algorithm 3 (presented in Section 4.4.3) to a Round Robin where clusters are selected one after the other in a cyclic manner.

Figure 4.15 presents the comparison of the two heuristics with 10 scenarios of 180 months each. The figure shows the gains of the repartition algorithm on the total execution time given in percents. All the values for the resources from 11 to 120 are used for this comparison (after that, the results would not change anymore because there would be too many resources available on each cluster). The heuristic used to obtain the grouping of resources on the clusters is the one using the Knapsack representation.

Gains on execution time increase with the number of resources per cluster. We observe a stepwise increase. Steps are due to fact that the repartition algorithm stops using some clusters while Round Robin continues to use all clusters. As we can see, there are 4 steps, each step occurring when one cluster stops being used by the repartition algorithm. When reaching 112 resources, the gain stays constant. In the figure, the gain is around 25%. This

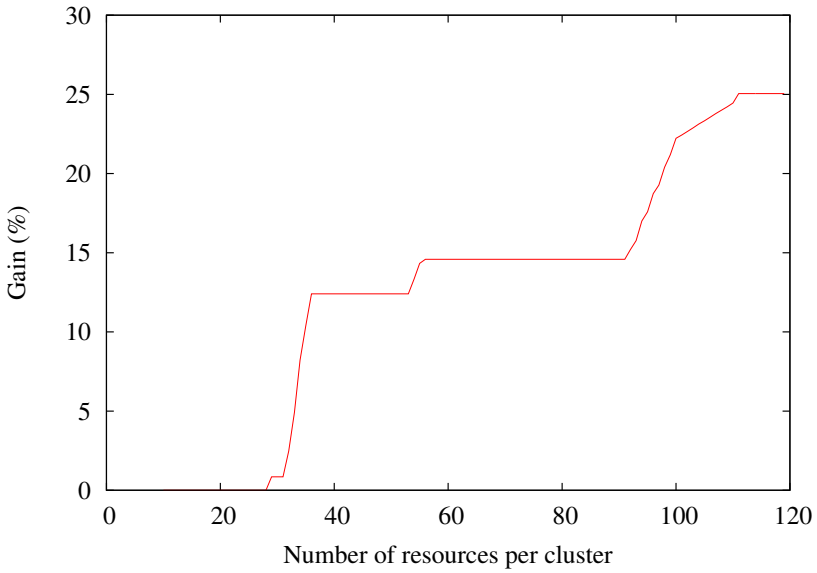


Figure 4.15 – Gain on execution time of the repartition heuristic compared to Round Robin on 10 scenarios.

corresponds exactly to the difference of execution time between the fastest and the slowest cluster when computing a one month simulation on 11 processors. It is normal because with enough resources, the total execution time of 1 or 10 scenarios is the same.

Figure 4.16 plots the comparison of the two algorithms, with more scenarios. The resources are increased by steps of 25. Even with an increase of the load, the behavior stays almost the same. Indeed, the steps are still present. However the duration of each step is linked to the number of scenarios. The more scenarios, the longer the steps. With a lot of resources, all executions eventually reach a gain of 25%. An interesting point showed in this figure is that with a lot of scenarios and few resources per cluster, there is a gain of around 10% which disappears when more resources are added before reappearing when more resources are available. This happens because with enough scenarios, changing the repartition will have a significant impact on the grouping of the processors and thus on the makespan.

The maximum gain of the repartition algorithm is easily predictable. Gains depend on the difference between the fastest and the slowest cluster. Thus the maximum expected gain corresponds to this difference. If clusters are almost homogeneous, the repartition algorithm acts almost as a Round Robin. However, instead of sending the scenarios on clusters one after another, it loads a cluster and then goes to the next. Reciprocally, if clusters are very heterogeneous, the expected gains are also very high.

Figure 4.17 shows the execution time of the simulations conducted with both algorithms. Up to 29 resources, both heuristics produce the same results, which is in accordance with the 0% gain in Figure 4.15. The steps are also visible. The figure shows clearly that with a few resources the algorithm used has no importance; however, with more resources,

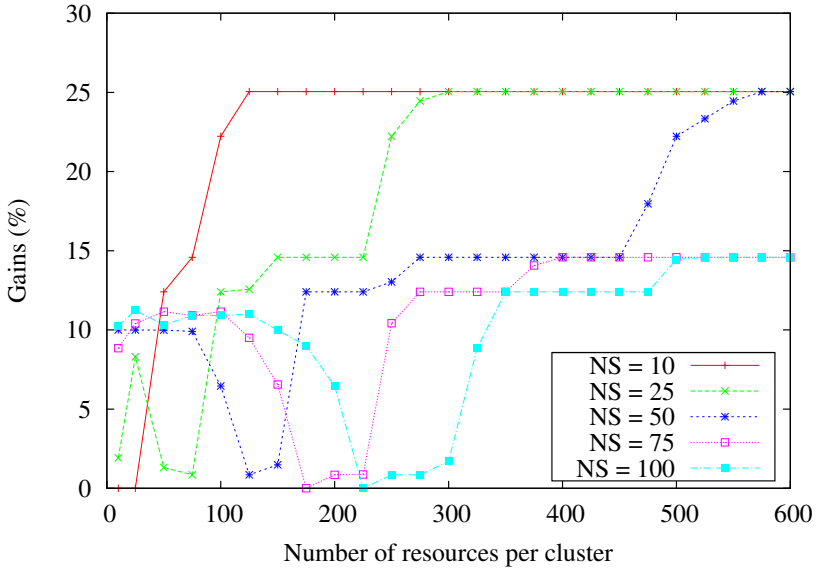


Figure 4.16 – Gains on the execution times of the repartition heuristic compared to Round Robin with a high number of scenarios.

the repartition algorithm decreases the execution time and the difference grows with the resources per cluster.

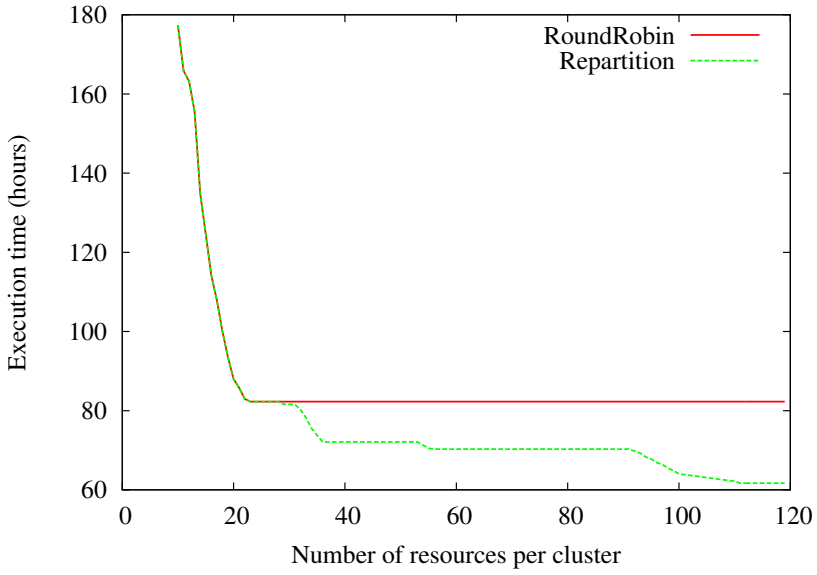


Figure 4.17 – Execution time obtained by the repartition heuristic and Round Robin on 10 scenarios.

Figure 4.18 presents the execution time obtained by the two algorithms for a high number of scenarios. To render the graph more readable, the execution time obtained for more than 25 scenarios with less than 50 resources are omitted as their values are out of the scale of the plot. With 10 resources per cluster and 100 scenarios, the gain is more than a week. So with few resources per cluster, using a smart repartition algorithm is important.

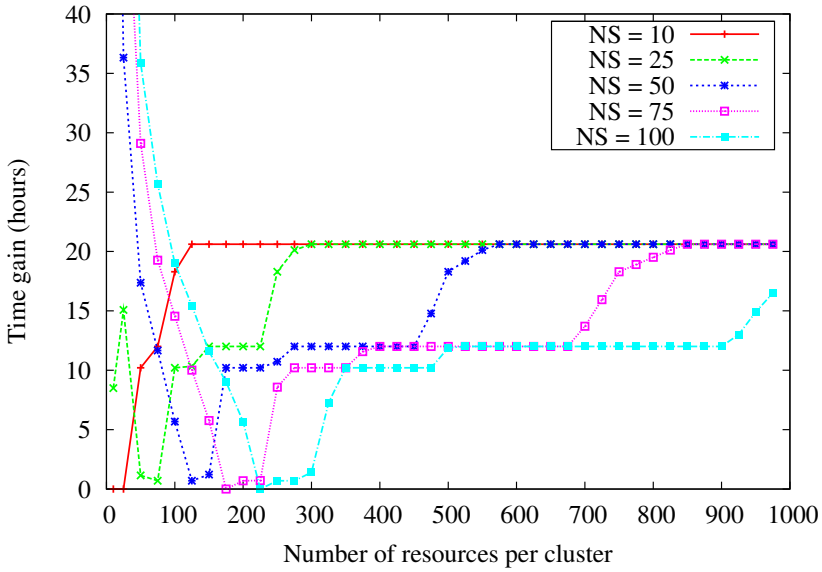


Figure 4.18 – Gains in hours obtained by the repartition heuristic compared to Round Robin with a high number of scenarios.

4.6 Execution on the Grid

This section is divided in three parts. The first part presents the implementation details that were necessary to perform the experiments. Then, we present a problem caused by MPI that arose during the experiments and the solutions we provided. Finally, we compare the real life executions with the simulations in order to determine the accuracy of the simulations.

4.6.1 Implementation

DIET provides a hierarchical scheduling in order to distribute the computations among the Local Agents (see Section 2.3.3 for details on DIET). In order to perform the repartition algorithm, the knowledge of all the SeDs (Servers Daemons executing the Ocean-Atmosphere service) is necessary. Indeed, the Algorithm presented in Section 4.4.3 needs all the performance evaluations before starting. The whole set of servers is known only at the top of the hierarchy by the MA and the client. In order not to slowdown the services

research done by the MA, we decided to implement the repartition algorithm within the client.

While the client is in charge of the global scheduling, the SeD is in charge of the local scheduling. The implementation done in the SeDs is divided in two parts: the performance evaluation and the service execution. The performance evaluation is launched when a request arrives and it executes the estimation provided by the Knapsack representation. Concerning the service execution, the SeD executes the set of scenarios it received, still by grouping the tasks using the Knapsack.

4.6.1.1 Checkpointing

In order to execute the application properly, we had to implement a basic fault tolerance mechanism in the SeDs. When a problem occurs during the execution, it must be possible to restart a scenario from the last month that has been properly executed. The application crashes on some occasions which are still unknown and relaunching the scenario, starting from the last month correctly executed, allows a proper execution. This method is a basic checkpointing mechanism.

An additional reason for this feature is that Grid'5000 is not a production grid. Each person can only reserve nodes for some time. In our case, the reservation is limited to 10 hours. Hence, executing an entire experiment of 10 scenarios each with 1800 months is impossible. The place is really limited so we have to relaunch the experiment several times to complete it. We launched an entire experiment on 5 clusters. If we could have all the resources needed and all the time, the execution would last about one month and a half. With the resources number and the reservations on Grid'5000, it took us more than 5 months to execute an entire experiment.

4.6.1.2 Multiple Asynchronous Call

The repartition algorithm needs a request corresponding to the whole set of scenarios and creates subsets that are sent to the SeDs for execution. To launch the execution of a service in DIET, the client has to use `diet_async_call()` provided in the API. This function sends the request to the DIET hierarchy and returns immediately. Internally, it starts by sending a request to have the list of all available SeDs and then performs the scheduling algorithm. Finally, it sends the corresponding partition of scenarios to the different SeDs.

`diet_async_call()` as defined in DIET can only send execution requests to one cluster at a time. So, we had to implement the possibility to make submissions to several clusters with just one call from the client. We decided to add this feature inside the implementation of the API. This could have been avoided, but we wanted the repartition to be done at the API level, not by the programmer of the client. Three reasons support this choice. First, if the client programmer does it, he has to access the DIET internal data, but he is not supposed to. Second, if it is in the API, the code can be re-used to implement another global scheduling algorithm for another application. Finally, doing it at the API level avoids the modification of the client code and is thus transparent for the client's

programmer.

Hence, to distribute DAGs among servers, we added a call to the repartition algorithm into `diet_async_call()`. Once the scheduling is done, the existing code of the function is executed several times to send each subset of scenarios, one subset for each selected cluster. This new functionality is now embedded in DIET.

Each request is given a unique ID on arrival. So, because the original request is divided into several requests after the ID assignation, each execution requested on a cluster has the same ID. This can disturb some existing tools around DIET. For example, VizDiet is a tool used to monitor the requests, so it is not supposed to have several requests with the same ID. A new API to simplify DIET internals is currently under development, so this problem will be easily corrected once the new API will be made available.

4.6.2 Revised Grouping Heuristic

4.6.2.1 Problem

A bug in OpenMPI (the MPI implementation used in ARPEGE) made us change the grouping algorithm: if 2 scenarios are executed on the same node of the Grid, the application behaves abnormally and both scenarios crash. All nodes on Grid'5000 are at least bi-cores, and there are clusters with bi-processors each with 2 cores so the problem can occur very often.

ARPEGE uses advanced features provided in OpenMPI, which are not always present in all MPI implementations. Thus, compiling with another MPI library is problematic. Furthermore, changing the MPI library changes the results of the application. Indeed, ARPEGE is very dependent of the MPI library for the results. If the messages in MPI are not exchanged in the same order due to some buffering, the results are different. Any small variation in some intermediary result can have a large effect on the final result. With this second problem, replacing the MPI library was made impossible, so we had to adapt to the technical difficulties and propose a new algorithmic solution.

4.6.2.2 Solution Leading to the Revised Grouping Heuristic

To avoid the bug, we added another constraint to the linear problem solved to obtain the grouping. The new constraint forces the number of processors in each group to be divisible by the number of cores of the nodes (2 or 4) thus avoiding to have several scenarios on a single node.

When working on clusters with 4 cores per node, we cannot use more than 8 processors for a single main-task because it needs between 4 and 11 processors. We benchmarked the application on 12 processors on a few clusters and it gave the same execution time as on 11. Thus, to allow the use of bigger groups of resources, we added the execution time on 12 processors, keeping the same value as the one on 11. This avoids a big slowdown of the application when working on a lot of resources.

The grouping changes so the makespan is also impacted. Figure 4.19 plots the average loss of time on 5 different clusters between the 2 extra-constrained Knapsack representation (with and without adding the 12^{th} processor) in regards to the original version. With a few

resources, both versions are bad. But, when the number of resources grows, the version with the 12^{th} processor behaves better. With 121 or more resources, this new version gives the same results as the original one. Instead of 112 resources to obtain the best makespan, the Knapsack version using 12 processors as maximum number of processors needs 121 processors (10 groups of 12 resources for the main-tasks and 1 to execute the post-processing tasks). The constrained version of the algorithm with 11 resources stays approximately 10% slower than the original version of the Knapsack algorithm after 111 resources. The constrained version can improve the makespan until 101 resources (10 groups of 10 resources and one for post-processing) and the original version improves the execution time until 111 resources, hence the stable difference between the two versions.

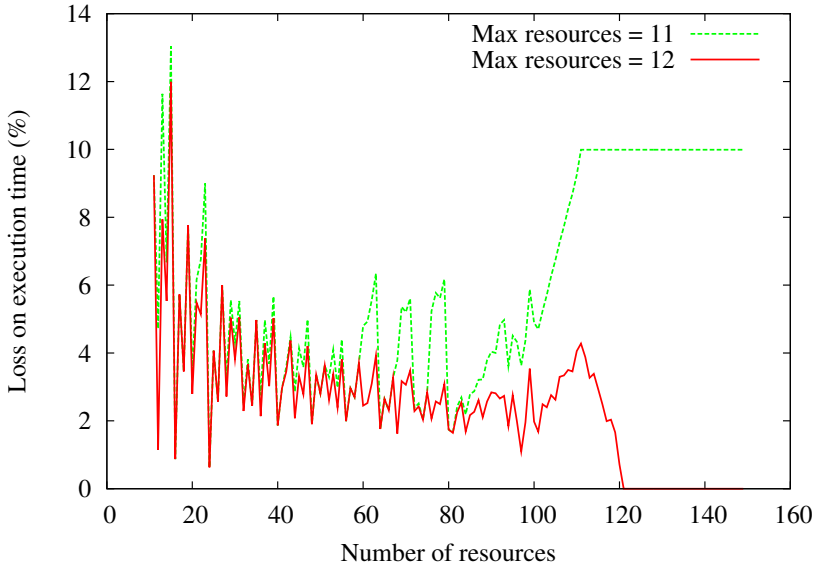


Figure 4.19 – Comparison between 3 versions of the Knapsack representation.

4.6.3 Real-life Experiments vs. Simulation Results

To test the accuracy of our simulations, we compared the simulated times of seven experiments with the real execution times. We did not use any resource to execute post-processing tasks during the main-tasks phase.

Figure 4.20 plots the differences of execution time between simulations and experiments. The positive difference means that the real-life experiment is slower than the simulation, and the negative means that is it faster. Experiments have been conducted on small scenarios in order to avoid the perturbations of the checkpointing phases.

The figure presents the differences for the whole experiment and for the main-tasks only. The difference between the simulations and the real execution time is sometimes large. Gathering of all the data for the post-processing tasks at the end is overloading the NFS and the performance really decreases. The model presented in Section 4.4 did not take

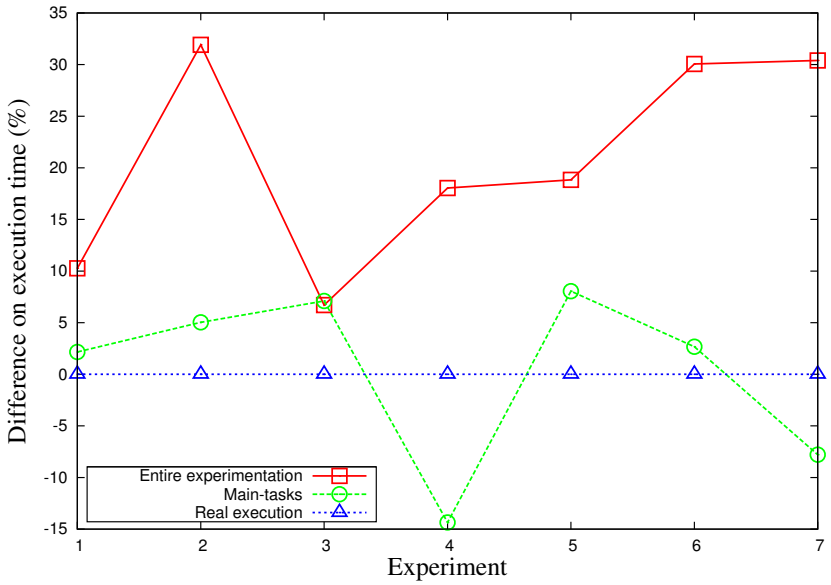


Figure 4.20 – Comparison between simulations and real-life experiments.

into account the contention to get the data. To avoid this problem, keeping a single resource to execute the post-processing tasks is sufficient.

When looking only at the differences for the main-tasks, results are better than for the whole experiments. Most of the time, the real computation time is less than 7% slower than the simulated time. In the fourth experience, it is 14% faster. In the seventh experiment, the real execution is also 7.8% faster. The average difference between simulations and the real experiments is 6.3%. This difference is quite good, but as we can see the results are very different from one experiment to another.

These results show that simulations really depend on the cluster load. The benchmarks were made with a specific cluster load, so if this load changes during the execution, the time will be different from the one expected. If other applications are heavily using the cluster network, there can be a non-negligible influence on the execution times of the application.

The slowdown due to the post-processing tasks is also well represented in this figure. In all experiments, the main-tasks time is quite good, but the total time is bad except in the third one. So, the post-processing tasks are really slowing the execution. The third experiment is good because the post-processing tasks where executed by groups of 4. When executed by groups of 4, the NFS used for the experiment is able to have the same performance that with only one post-processing task. The average difference between simulations and real-life experiments is 20.8% on the total execution time. This difference is big and still has to be improved to allow the simulations to be as precise as possible. A model taking contentions in data transmissions would help solving this problem.

4.7 Summary

In this chapter we have discussed the problem of scheduling a specific application on an heterogeneous Grid composed of several clusters. The application is a real life simulator for climate prediction using different models to be as accurate as possible. The application is launched several times in parallel on the Grid in order to have a better idea of the influencing parameters on the results of the climate simulations.

From the model of the application that was designed, we provide different scheduling heuristics to execute simulations in parallel on a single homogeneous cluster. The heuristics create groups of processors that are dedicated to the execution of the main parallel task of the application. We compare these heuristics, and results show that the Knapsack representation of the problem should be used to obtain the best execution time.

On top of this local scheduling, we add another layer that schedules the application globally onto a Grid. Clusters on this Grid can be heterogeneous. The algorithm we propose takes the whole set of simulations and divides it into several subsets that will be executed on different clusters. We compared this algorithm to a Round Robin and showed that it was better to reduce the overall makespan. The maximum gain corresponds to the difference between the fastest and the slowest cluster.

Both these scheduling solutions were implemented within the DIET GridRPC middleware. The local scheduling is done by a service running on each cluster. Using the dynamic information provided by the servers, the client can then compute the repartition on the different clusters. The implementation led to new features added to DIET. However, technical difficulties forced us to adapt the local scheduling policy by adding more constraints, thus impacting the execution time.

Because we adapted the local scheduling policy, we compared the original version of the heuristic and the modified one. If enough resources are used, the results are the same. However, on a small number of resources a loss of a few percents on the execution time is caused. Finally, we compared the simulations with the real life experiments. Results show that simulations taking only the main task of the application into account are quite accurate, but if the small post-processing tasks are taken into account, the simulations diverge from the reality.

CHAPTER 5

Conclusions and Perspectives

Contents

This chapter concludes the Thesis. It summarizes our contributions on the different topics and gives some possible directions for future work.

We studied the problem of scheduling and managing jobs in computational Grids in two different contexts. First, we studied a reallocation mechanism, which moves waiting jobs between clusters. We showed that this is beneficial for users. Furthermore, our solution is easy to deploy because it does not require to modify existing platforms. Secondly, we designed and compared heuristics to schedule a climatology application over a Grid and we presented the implementation in DIET *to run the application over* GRID'5000.

Main directions of future works are to extend the work presented in the Thesis. On tasks reallocation, implementing the reallocation mechanism in the DIET *grid middleware is one objective. Another possible continuation is to study different policies to choose automatically the number of processors for moldable jobs. Concerning the scheduling for Ocean-Atmosphere, it would be interesting to extend the heuristics to a more general class of* DAG*s as well as studying the scheduling in a non-dedicated environment.*

5.1 Summary and Conclusions

Scientific researchers in many fields turn to computer science and High Performance Computing to solve problems or perform simulations. They develop increasingly complex models that always require more computational power. While the increase in processor speed fulfilled their needs for many years, nowadays, the only way to increase the available computing power is to group resources. The Grid is the aggregation of geographically distributed and heterogeneous resources. While it enables scientists to have more processing power at their disposal, accessing a Grid is a very complex matter. To tackle this issue, grid middleware have been developed to hide the complexity of accessing such resources.

The work presented in this Thesis focuses on the scheduling of parallel applications in Grid environments composed of multiple parallel machines or clusters. As presented in the introduction, the problems related to scheduling in Grid environments are numerous. We focused on jobs scheduling and resource management done at the middleware level. The middleware has several issues to assess. Among other it must be able to communicate through the network, take the resource heterogeneity into account, communicate with different pieces of software, recover from runtime errors such as disappearing resources, provide efficient scheduling, *etc*.

In order to give the necessary background to understand the Thesis, Chapter 2 introduced all necessary concepts. We started by presenting the evolution of the hardware architectures of parallel and distributed systems. New distributed computing platforms rely on the extension of existing architectures. Therefore we presented the different architectures, starting with parallel machines, extending it to clusters, and then Grids, to finish with Cloud and Sky Computing. Scheduling is a very large problem, thus we presented some problems and existing solutions to scheduling in Grids. Scheduling can be done at different levels. It can be meta-scheduling at the global level as well as local scheduling within batch schedulers. Finally, we presented actual Grids, middleware, batch schedulers, and tools used in practice so that the reader understands how the work presented here is used in a real environment.

The first focus of our work, presented in Chapter 3, was to design a generic and dynamic middleware solution to overcome scheduling errors in Grid systems. Each cluster in the considered environment is managed independently by its own batch scheduler and the middleware that runs on top of this architecture. Batch schedulers base their scheduling decisions on the runtime estimates of a job which are often very unreliable. Therefore, when a job finishes earlier than expected, the local scheduler modifies the schedule and the middleware must detect the change and adapt the global schedule. The solution we give is based on the GridRPC standard and can thus be implemented in any existing middleware compliant with the GridRPC API. To facilitate the implementation in DIET, we give directions on how to implement this reallocation mechanism.

In order to react to scheduling changes in the platform, we propose two reallocation mechanisms that can automatically move waiting jobs from one queue to another. Each reallocation algorithm is coupled to different scheduling heuristics. The middleware reallocates jobs between clusters when it detects that it would be beneficial to do so. In order to validate the solution, we performed simulations using real-life workload traces. We studied

different traces on different platforms with different kinds of tasks. We considered rigid and moldable jobs. The results show that the expected gain on the average job response time can be very substantial, even with the simplest algorithm. Indeed, the average response time of jobs is usually diminished by 5% to 20%. In the best cases, jobs spend twice less time in the platform on average. Thus, when using a Grid middleware, reallocation should be implemented to improve quality of service provided to users.

Chapter 4 presents the second part of the main focus of this Thesis. We worked on the design and implementation of scheduling heuristics for a climatology application in order to execute it efficiently over Grids. To provide efficient scheduling heuristics, we started by modeling the application as a simple DAG composed of a chain of identical moldable tasks followed by small sequential post-processing tasks. The target setup we aim at is the concurrent execution of those DAGs to perform the climate simulations as fast as possible. To execute the simulations in parallel, we use a Grid and divide the scheduling in two levels. At the Grid level, the simulations are distributed among different clusters. Then, on each cluster, the resources are divided among the simulations.

We compared different scheduling policies at the cluster level and showed that the Knapsack representation is the best one. It provides the best results and is fast to execute. To evaluate the distribution among clusters, we compared our results with a simple round robin technique and showed that our repartition algorithm is better. Because both techniques give good results, we implemented them into the DIET middleware and performed experiments on GRID'5000 in order to test our scheduling policies. The execution on GRID'5000 had some technical problems, thus we modified the heuristics in order to handle the problems. We evaluated the loss of adding restrictions on our heuristics and showed that, with enough resources, the loss disappeared. Finally, we compared the difference between the estimation of the execution time of the application based on our model and the real execution on the Grid. Results show that, if only the moldable tasks are taken into account, the estimation is precise. However if the post-processing tasks are taken into account, errors arise.

In this Thesis we showed that extending a grid middleware allows a better management of jobs. The two solutions we give are based on the same idea. An agent running on top of the platform takes scheduling decisions based on runtime estimations done locally on the computational resources. The computations are thus done at two levels. Complex computations are done in parallel on each of the resources. Result from these computations are then used by a central agent to take the scheduling decisions. Using our solutions in real Grids will increase the satisfaction of users. Indeed, by providing an efficient and automatic solution, we are able to improve the global efficiency of the platform. Furthermore, deploying such solutions is easy because it does not require to modify the underlying architecture of the Grid. The grid middleware is deployed on top of resources and manages everything itself. Our solutions are feasible and efficient, therefore they should be used when possible.

5.2 Perspectives

The works we presented in this Thesis can be extended further in several directions.

Concerning tasks reallocation in a Grid environment, some work could be done on the

way moldable tasks are handled. In the current version of the management of moldable tasks, each task is considered separately without consideration for any other task. Thus, when the resources are free, the first task to enter gets everything. It may be more efficient to keep some free resources, so that when other tasks are submitted they also have resources as soon as they arrive. Giving half the resources to two tasks at the same time is more efficient than to give all resources to each task one after the other. Indeed, the speedup rate decreases with more resources, so the throughput is better when giving less resources to more tasks.

Another extension about the management of moldable jobs on a cluster could be to reserve resources through a batch scheduler and manage several jobs in a single reservation. Thus, the middleware could take several applications into account to determine the best size for the moldable jobs. The middleware could use a Knapsack representation of the problem to schedule the jobs inside the reservation. Several reservations could be submitted on the same batch scheduler, therefore instead of just choosing on what server to execute the application, it would be necessary to also choose the reservation slot.

It will also be interesting to study what happens when the middleware is able to predict the walltime with different levels of accuracy. On a dedicated platform, if the predictions are perfect, reallocation will never be performed. However, in a non-dedicated environment, even if the predictions of the middleware are perfect, reallocation could still come from errors introduced by other users. The other extreme would be to study what happens when a walltime prediction is impossible. In that case, the middleware would submit the jobs with the longest possible walltime. Therefore, the meta-scheduling decisions taken upon submission would be a basic load balancing. The middleware would then reallocate jobs dynamically as some of them finish their execution.

The previous point only take into account the case when the walltime is longer that the runtime. However, the middleware should also be able to finish a job when the estimation has been wrong and the walltime is reached before the end of the computation. A possible solution is to embed a checkpointing mechanism in the middleware and when the walltime of a job will be reached, trigger this mechanism and reallocate the job. Once reallocated (or resubmitted locally) the job can finish its execution.

From a more practical point of view, the next step is the implementation of the reallocation mechanism within the DIET middleware. As we presented in Section 3.3.3, DIET already provides most of the required features. Both reallocation algorithms will be implemented with MCT. The regular algorithm will be used on non-dedicated platforms. The all-cancellation mechanism will be implemented because DIET can be used in a dedicated environment. Furthermore, we could use this in the SPADES [1] project where we plan to maintain a set of reserved resources on each site which are managed by our own embedded batch schedulers.

Although the scheduling policies and implementation in DIET we provided for the execution of the climatology application reached their objectives, this work could also be extended in several directions.

In this study, we only considered clusters with a dedicated access. However on a real

1. ANR Project 08-ANR-SEGI-025

platform, the resources are shared between multiple users or organizations. The most common way to share resources is to use a batch scheduler. Thus, a possible continuation of this work is to take into account the resource sharing on a cluster. Using tools such as Simbatch, we can determine for how long and how many resources can be obtained at a given time. Then, with this information, the schedule can be adapted dynamically. In case of a large modification in the schedule, jobs reallocation as defined in the first part of the Thesis could be used to move the application between clusters.

Another extension of this work is the definition of a more generic scheduling heuristic to execute applications having a similar behavior as the one in our work. Indeed, executing several instances of the same DAG where each moldable sub-task of the DAG repeats itself with different input parameters is a pattern that can be found on all kind of applications. The model and heuristics we designed in the work could be extended so that a larger class of DAGs would benefit from the specialized scheduling heuristics.

Bibliography

[1] Amazon Elastic Compute Cloud, 2010. http://aws.amazon.com/ec2/. 17, 18

[2] Animoto , 2010. http://animoto.com/. 17

[3] EGI, European Grid Initiative, 2010. http://www.egi.eu. 28

[4] Google App Engine, 2010. http://www.google.com/apps. 17

[5] GÉANT2 Network, 2010. http://www.geant2.net/. 29

[6] Microsoft Azure, 2010. http://www.microsoft.com/windowsazure/. 17

[7] NAS parallel benchmark, 2010. http://www.nas.nasa.gov/Resources/Software/npb.html. 56

[8] Renater Network, 2010. http://www.renater.fr. 29

[9] The FutureGrid Project, 2010. http://futuregrid.org. 29

[10] The Network Simulator (ns2), 2010. http://nsnam.isi.edu/nsnam/. 33

[11] The OneLab Project, 2010. http://www.onelab.eu/index.php. 29

[12] The Parallel Workload Archive, 2010. http://www.cs.huji.ac.il/labs/parallel/workload/. 45

[13] The PlanetLab Project, 2010. http://www.planet-lab.eu. 29

[14] The TeraGrid Project, 2010. https://www.teragrid.org. 28

[15] The DEISA Project, 2010. http://www.deisa.eu/. 28

[16] Tivoli Software, 2010. http://www.ibm.com/software/tivoli/. 27

[17] Top 500 supercomputers, 2010. http://www.top500.org. 14

[18] Narasimha R. Adiga, Matthias A. Blumrich, Dong Chen, Paul Coteus, Alan Gara, Mark E. Giampapa, Philip Heidelberger, Sarabjeet Singh, Burkhard D. Steinmacher-Burow, Todd Takken, Mickey Tsao, and Pavlos Vranas. Blue Gene/L torus interconnection network. *IBM J. Res. Dev.*, 49(2):265–276, 2005. 15

[19] Abdelkader Amar, Raphaël Bolze, Yves Caniou, Eddy Caron, Benjamin Depardon, Jean-Sébastien Gay, Gaël Le Mahec, and David Loureiro. Tunable Scheduling in a GridRPC Framework. *Concurrency & Computation: Practice & Experience*, 20:1051–1069, 2008. 5

[20] Morris Jette And 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. 27

[21] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010. 17

[22] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009. 17

[23] Olaf Arndt, Bernd Freisleben, Thilo Kielmann, and Frank Thilo. A comparative study of online scheduling algorithms for networks of workstations. *Cluster Computing*, 3(2):95–112, 2000. 20

[24] Mark Baker, Geoffrey C. Fox, Hon W. Yau, and Tony Hey. A Review of Commercial and Research Cluster Management Software. Technical report, 1995. 23

[25] Sanjeev Baskiyar and Kiran Kumar Palli. Low power scheduling of DAGs to minimize finish times. *High Performance Computing-HiPC 2006*, pages 353–362, 2006. 22

[26] William F. Baxter, Robert G. Gelinas, James M. Guyer, Dan R. Huck, Michael F. Hunt, David L. Keating, Jeff S. Kimmell, Phil J. Roux, Liz M. Truebenbach, Rob P. Valentine, Pat J. Weiler, Joseph Cox, Barry E. Gillott, Andrea Heyda, Rob J. Pike, Tom V. Radogna, Art A. Sherman, Michael Sporer, Doug J. Tucker, and Simon N. Yeung. Symmetric multiprocessing computer with non-uniform memory access architecture, March 1999. 13

[27] Albeaus Bayucan, Robert L. Henderson, Casimir Lesiak, Bhroam Mann, Thomas Proett, and Dave Tweten. Portable batch system: External reference specification. Technical report, MRJ Technology Solutions, November 1999. 5

[28] Olivier Beaumont, Arnaud Legrand, Loris Marchal, and Yves Robert. Steady-state scheduling on heterogeneous clusters: why and how? In *6th Workshop on Advances in Parallel and Distributed Computational Models (APDCM)*. IEEE Computer Society Press, 2004. 22

[29] William H. Bell, David G. Cameron, A. Paul Millar, Luigi Capozza, Kurt Stockinger, and Floriano Zini. Optorsim: A Grid Simulator for Studying Dynamic Data Replication Strategies. *International Journal of High Performance Computing Applications*, 17(4):403–416, November 2003. 33

[30] Marta Beltrán and Antonio Guzmán. The Impact of Workload Variability on Load Balancing Algorithms. *Scalable Computing: Practice and Experience*, 10(2):131–146, June 2009. 36

[31] Michael Bender, Soumen Chakrabarti, and Sambavi Muthukrishnan. Flow and Stretch Metrics for Scheduling Continuous Job Streams. In *In Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 270–279, 1998. 22

[32] Anne Benoit, Loris Marchal, Jean-François Pineau, Yves Robert, and Frédéric Vivien. Offline and online scheduling of concurrent bags-of-tasks on heterogeneous platforms. In *10th Workshop on Advances on Parallel and Distributed Processing Symposium (APDCM 2008)*. IEEE Computer Society, 2008. 4, 22

[33] Rémi Bertin, Arnaud Legrand, and Corinne Touati. Toward a fully decentralized algorithm for multiple bag-of-tasks application scheduling on grids. Research Report 6537, INRIA, May 2008. 26

[34] Raphaël Bolze, Franck Cappello, Eddy Caron, Michel Daydé, Frederic Desprez, Emmanuel Jeannot, Yvon Jégou, Stéphane Lanteri, Julien Leduc, Noredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, Benjamin Quetier, Olivier Richard, El-Ghazali Talbi, and Irena Touché. Grid'5000: A Large Scale and Highly Reconfigurable Experimental Grid Testbed. *International Journal of High Performance Computing Applications*, 20(4):481–494, 2006. 29

[35] Raphaël Bolze. *Analyse et déploiement de solutions algorithmiques et logicielles pour des applications bioinformatiques à grande échelle sur la grille*. PhD thesis, École Normale Supérieure de Lyon, October 2008. 43

[36] Robert Bradford, Evangelos Kotsovinos, Anja Feldmann, and Harald Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, pages 169–179, New York, NY, USA, 2007. ACM. 36

[37] Tracy D. Braun, Howard Jay Siegel, Noah Beck, Lasislau L. Bölöni, Muthucumara Maheswaran, Albert I. Reuther, James P. Robertson, Mitchell D. Theys, Bin Yao, Debra Hensgen, and Richard F. Freund. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 61(6):810–837, 2001. 4, 26

[38] Jérémy Buisson, Ozan Sonmez, Hasim Mohamed, Wouter Lammers, and Dick Epema. Scheduling Malleable Applications in Multicluster Systems. Technical Report TR-0092, Institute on Resource Management and Scheduling, CoreGRID - Network of Excellence, May 2007. 5

[39] Rajkumar Buyya. *High Performance Cluster Computing: Architectures and Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999. 13

[40] Rajkumar Buyya and Manzur Murshed. Gridsim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing. *CoRR*, cs.DC/0203019, 2002. 33

[41] Rajkumar Buyya, Rajiv Ranjan, and Rodrigo Calheiros. InterCloud: Utility-Oriented Federation of Cloud Computing Environments for Scaling of Application Services. In Ching-Hsien Hsu, Laurence Yang, Jong Park, and Sang-Soo Yeo, editors, *Algorithms and Architectures for Parallel Processing*, volume 6081 of *Lecture Notes in Computer Science*, pages 13–31. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-13119-6_2. 18

[42] Murray Campbell, A. Joseph Hoane, and Feng-hsiung Hsu. Deep Blue. *Artificial Intelligence*, 134(1-2):57 – 83, 2002. 15

[43] Yves Caniou, Eddy Caron, Ghislain Charrier, Andréea Chis, Frédéric Desprez, and Eric Maisonnave. Ocean-Atmosphere Modelization over the Grid. In Wu-chi Feng

and Yuanyuan Yang, editors, *The 37th International Conference on Parallel Processing (ICPP 2008)*, pages 206–213, Portland, Oregon. USA., September 9-11 2008. IEEE. 70

[44] Yves Caniou, Eddy Caron, Ghislain Charrier, Andréea Chis, Frédéric Desprez, and Éric Maisonnave. Ocean-Atmosphere Modelization over the Grid. Technical Report RR-6695, Institut National de Recherche en Informatique et en Automatique (INRIA), October 2008. Also available as LIP Research Report RR-2008-26. 70

[45] Yves Caniou, Eddy Caron, Ghislain Charrier, and Frédéric Desprez. Meta-Scheduling and Task Reallocation in a Grid Environment. In *The Third International Conference on Advanced Engineering Computing and Applications in Sciences (ADVCOMP'09)*, pages 181–186, Sliema, Malta, October 11-16 2009. 36, 41, 48

[46] Yves Caniou, Eddy Caron, Ghislain Charrier, Frédéric Desprez, Éric Maisonnave, and Vincent Pichon. Ocean-Atmosphere Application Scheduling within DIET. In *APDCT-08 Symposium. International Symposium on Advanced in Parallel and Distributed Computing Techniques*, pages 675–680, Sydney. Australia., December 10-12 2008. In conjunction with ISPA'2008, IEEE Computer Society. Invited paper from the reviewed process of ISPA'08. 70

[47] Yves Caniou, Eddy Caron, Ghislain Charrier, Frédéric Desprez, Éric Maisonnave, and Vincent Pichon. Ocean-Atmosphere Application Scheduling within DIET. Technical Report RR-6836, Institut National de Recherche en Informatique et en Automatique (INRIA), February 2009. 70

[48] Yves Caniou, Ghislain Charrier, and Frédéric Desprez. Analysis of Tasks Reallocation in a Dedicated Grid Environment. In *IEEE International Conference on Cluster Computing 2010 (Cluster 2010)*, pages 284–291, Heraklion, Crete, Greece, September 20-24 2010. 36

[49] Yves Caniou, Ghislain Charrier, and Frédéric Desprez. Analysis of Tasks Reallocation in a Dedicated Grid Environment. Technical Report RR-7226, Institut National de Recherche en Informatique et en Automatique (INRIA), March 2010. 36, 48, 49

[50] Yves Caniou, Ghislain Charrier, and Frédéric Desprez. Evaluation of Reallocation Heuristics for Moldable Tasks in Computational Dedicated and non Dedicated Grids. Technical Report RR-7365, Institut National de Recherche en Informatique et en Automatique (INRIA), August 2010. 36

[51] Yves Caniou, Ghislain Charrier, and Frédéric Desprez. Evaluation of Reallocation Heuristics for Moldable Tasks in Computational Grids. In ACM, editor, *9th Australasian Symposium on Parallel and Distributed Computing (AusPDC 2011)*, page 10, Perth, Australia, January 17-20 2011. To appear. 36

[52] Yves Caniou and Jean-Sébastien Gay. Simbatch: An API for simulating and predicting the performance of parallel resources managed by batch systems. In *Workshop on Secure, Trusted, Manageable and Controllable Grid Services (SGS), held in conjunction with EuroPar'08*, volume 5415 of *LNCS*, pages 217–228, 2009. 4, 34

[53] Nicolas Capit, Georges Da Costa, Yiannis Georgiou, Guillaume Huard, Cyrille Martin, Grégory Mounié, Pierre Neyron, and Olivier Richard. A Batch Scheduler with High Level Components. *CoRR*, abs/cs/0506006:9, 2005. 27

[54] Eddy Caron. *Contribution to the management of large scale platforms: the DIET experience*. HDR (Habilitation à Diriger les Recherches), École Normale Supérieure de Lyon, October 2010. 31, 43

[55] Eddy Caron, Andréea Chis, Frédéric Desprez, and Alan Su. Design of plug-in schedulers for a GridRPC environment. *Future Generation Computer Systems*, 24(1):46–57, January 2008. 43

[56] Eddy Caron, Frédéric Desprez, Frédéric Lombard, Jean-Marc Nicod, Martin Quinson, and Frédéric Suter. A scalable approach to network enabled servers. In B. Monien and R. Feldmann, editors, *Proceedings of the 8th International EuroPar Conference*, volume 2400 of *Lecture Notes in Computer Science*, pages 907–910, Paderborn, Germany, August 2002. Springer-Verlag. 26

[57] Eddy Caron, Frédéric Desprez, and Gaël Le Mahec. DAGDA: Data Arrangement for the Grid and Distributed Applications. In *ESCIENCE '08: Proceedings of the 2008 Fourth IEEE International Conference on eScience*, pages 680–687, Washington, DC, USA, 2008. IEEE Computer Society. 44

[58] Eddy Caron and Frédéric Desprez. DIET: A Scalable Toolbox to Build Network Enabled Servers on the Grid. *International Journal of High Performance Computing Applications*, 20(3):335–352, 2006. 31

[59] Eddy Caron, Vincent Garonne, and Andreï Tsaregorodtsev. A study of meta-scheduling architectures for high throughput computing. 2005-13, Laboratoire de l'Informatique du Parallélisme (LIP), Lyon, France, May 2005. 25

[60] Henri Casanova, Frédéric Desprez, and Frédéric Suter. On Cluster Resource Allocation for Multiple Parallel Task Graphs. Technical report. 75

[61] Henri Casanova, Frédéric Desprez, and Frédéric Suter. Minimizing Stretch and Makespan of Multiple Parallel Task Graphs via Malleable Allocations. *Parallel Processing, International Conference on*, 0:71–80, 2010. 5

[62] Henri Casanova and Jack Dongarra. NetSolve: a Network Server for Solving Computational Science Problems. In *Supercomputing'96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, page 40, Washington, DC, USA, 1996. IEEE Computer Society. 31

[63] Henri Casanova, Arnaud Legrand, and Martin Quinson. Simgrid: a Generic Framework for Large-Scale Distributed Experiments. In *10th IEEE International Conference on Computer Modeling and Simulation*, March 2008. 33

[64] Henri Casanova, Dmitrii Zagorodnov, Francine Berman, and Arnaud Legrand. Heuristics for Scheduling Parameter Sweep Applications in Grid Environments. In *HCW '00: Proceedings of the 9th Heterogeneous Computing Workshop*, page 349, Washington, DC, USA, 2000. IEEE Computer Society. 19

[65] Ghislain Charrier. From Scheduling Theory to Practice: a Case Study. In *The 5th International Conference on Soft Computing as Transdisciplinary Science and Technology*, Cergy-Pontoise. France, October 28-31 2008. ACM/IEEE. 70

[66] Ghislain Charrier and Yves Caniou. Ordonnancement et réallocation de tâches sur une grille de calcul. In *19emes Rencontres francophonnes du Parallélisme (RenPar'19)*, Toulouse, France, Septembre 9-11 2009. 36

[67] Walfredo Cirne and Francine Berman. Using Moldability to Improve the Performance of Supercomputer Jobs. *Journal of Parallel and Distributed Computing*, 62(10):1571–1601, 2002. 56

[68] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, 1998. 12

[69] Sivarama P. Dandamudi. *Hierarchical Scheduling in Parallel and Cluster Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 2003. 4

[70] T. Decker, T. Lücking, and B. Monien. A 5/4-approximation algorithm for scheduling identical malleable tasks. *Theor. Comput. Sci.*, 361(2):226–240, 2006. 19

[71] E. Dekel and S. Sahni. Parallel scheduling algorithms. *Operations Research*, 31(1):24–49, 1983. 19

[72] Frédéric Desprez and Frédéric Suter. A Bi-criteria Algorithm for Scheduling Parallel Task Graphs on Clusters. *Cluster Computing and the Grid, IEEE International Symposium on*, 0:243–252, 2010. 20

[73] Fangpeng Dong and Selim G. Akl. Scheduling algorithms for grid computing: State of the art and open problems. Technical report, School of Computing, January 2006. 26

[74] Hevé Douville. Validation and sensitivity of the global hydrologic budget in standalone simulations with the isba land surface scheme. *Clim Dyn*, 14:151–171, 1998. 71

[75] Pierre-Francois Dutot, Lionel Eyraud, Grégory Mounié, and Denis Trystram. Bicriteria algorithm for scheduling jobs on cluster platforms. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 125–132. ACM, 2004. 4

[76] Michel Déqué, Christine Dreveton, Alain Braun, and Daniel Cariolle. The ARPEGE/IFS atmosphere model: a contribution to the french community climate modeling. *Clim Dyn*, 10:249–266, 1994. 71

[77] Constantinos Evangelinos and Chris N. Hill. Cloud Computing for Parallel Scientific HPC Applications: Feasibility of Running Coupled Atmosphere-Ocean Climate Models on Amazon's EC2. In *Cloud Computing and Its Applications (CCA-08)*, October 2008. 18

[78] Dror G. Feitelson and Ahuva W. Mu'alem. Utilization and predictability in scheduling the ibm sp2 with backfilling, 1998. 24

[79] Dror G. Feitelson and Larry Rudolph. Parallel job scheduling: Issues and approaches, 1995. 19

[80] Dror G. Feitelson and Larry Rudolph. Metrics and benchmarking for parallel job scheduling. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing (IPDPS/JSSPP '98)*, pages 1–24, London, UK, 1998. Springer-Verlag. 4, 21, 22, 47

[81] Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Parallel job scheduling - a status report. In *Job Scheduling Strategies for Parallel Processing*, 2004. 23

[82] Dror G. Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C. Sevcik, and Parkson Wong. Theory and Practice in Parallel Job Scheduling. In *IPPS '97: Proceedings of the Job Scheduling Strategies for Parallel Processing*, pages 1–34, London, UK, 1997. Springer-Verlag. 3, 19

[83] Dror G. Feitelson and Dan Tsafrir. Workload Sanitation for Performance Evaluation. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 221–230, Austin, Texas, March 2006. 46

[84] Thierry Fichefet and Miguel Ángel Morales Maqueda. Sensitivity of the global sea ice model to the treatment of ice thermodynamics. *J Geophys Res*, 102:C6:12609–12646, 1997. 71

[85] Ian Foster. What is the Grid? - a three point checklist. *GRIDtoday*, 1(6), July 2002. 16

[86] Ian Foster. Globus toolkit version 4: Software for service-oriented systems. *Journal of Computer Science and Technology*, 21(4):513–520, July 2006. 30

[87] Ian Foster and Carl Kesselman, editors. *The Grid: blueprint for a new computing infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999. 16

[88] Ian Foster, Carl Kesselman, and Steven Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Int. J. High Perform. Comput. Appl.*, 15(3):200–222, 2001. 16

[89] Ian Foster, Yong Zhao, Ioan Raicu, and Shiyong Lu. Cloud Computing and Grid Computing 360-Degree Compared. In *2008 Grid Computing Environments Workshop*, pages 1–10. IEEE, November 2008. 17

[90] Richard F. Freund, Michael Gherrity, Stephen Ambrosius, Mark Campbell, Mike Halderman, Debra Hensgen, Elaine Keith, Taylor Kidd, Matt Kussow, John D. Lima, Francesca Mirabile, Lantz Moore, Brad Rust, and Howard Jay Siegel. Scheduling resources in multi-user, heterogeneous, computing environments with smartnet. In *7th IEEE Heterogeneous Computing Workshop (HCW '98)*, pages 184–199, 1998. 4

[91] Pedro Garcia, Carles Pairot, Ruben Mondejar, Jordi Pujol, Helio Tejedor, and Robert Rallo. *PlanetSim: A New Overlay Network Simulation Framework*. 2005. 33

[92] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979. 19

[93] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel virtual machine: a users' guide and tutorial for networked parallel computing*. MIT Press, Cambridge, MA, USA, 1994. 14

[94] Wolfgang Gentzsch. Sun Grid Engine: Towards Creating a Compute Power Grid. *Cluster Computing and the Grid, IEEE International Symposium on*, 0:35, 2001. 5

[95] Brent Gorda and Richard Wolski. Time sharing massively parallel machines. In *International conference on parallel processing*, pages 214–217, August 1995. 22

[96] Ricardo Graciani-Diaz, Andrei Tsaregorodtsev, and Adria Casajus Ramo. Pilot Framework and the DIRAC WMS. In *CHEP 09*, May 2009. 5

[97] William Gropp and Ewing Lusk. A taxonomy of programming models for symmetric multiprocessors and SMP clusters. In *PMMP '95: Proceedings of the conference on Programming Models for Massively Parallel Computers*, page 2, Washington, DC, USA, 1995. IEEE Computer Society. 12

[98] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*. MIT Press, Cambridge, MA, USA, 1994. 14

[99] Francesc Guim and Julita Corbalán. A Job Self-scheduling Policy for HPC Infrastructures. In *Job Scheduling Strategies for Parallel Processing (JSSPP)*, 2008. 37

[100] Eldon C. Hall. *Journey to the Moon: The History of the Apollo Guidance Computer*. American Institute of Aeronautics and Astronautics (AIAA), 1996. ISBN 156347185X. 2

[101] Leslie A. Hall, David B. Shmoys, and Joel Wein. Scheduling to minimize average completion time: off-line and on-line algorithms. In *SODA '96: Proceedings of the seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 142–151, Philadelphia, PA, USA, 1996. Society for Industrial and Applied Mathematics. 20

[102] Janko Heilgeist, Thomas Soddemann, and Harald Richter. Design and Implementation of a Distributed Metascheduler. In *The First International Conference on Advances in P2P Systems (AP2PS 2009)*, pages 63–72, Sliema, Malta, October 11-16 2009. 26

[103] Kuo-Chan Huang, Po-Chi Shih, and Yeh-Ching Chung. Adaptive Processor Allocation for Moldable Jobs in Computational Grid. *International Journal of Grid and High Performance Computing*, 1(1):10–21, January-March 2009. 38

[104] Alexandru Iosup, Mathieu Jan, Ozan Sonmez, and Dick H. J. Epema. On the Dynamic Resource Availability in Grids. In *GRID '07: Proceedings of the 8th IEEE/ACM International Conference on Grid Computing*, pages 26–33, Washington, DC, USA, 2007. IEEE Computer Society. 16

[105] Michael A. Iverson and Füsun Özgüner. Hierarchical, competitive scheduling of multiple DAGs in a dynamic heterogeneous environment. *Distributed Systems Engineering*, 6:112–120, September 1999. 5

[106] Heath A. James, Ken A. Hawick, and Paul D. Coddington. Scheduling Independent Tasks on Metacomputing Systems. In *in Proceedings of Parallel and Distributed Computing Systems*, 1999. 25

[107] Márk Jelasity, Alberto Montresor, Gian Paolo Jesi, and Spyros Voulgaris. The Peersim Simulator. `http://peersim.sf.net`. 33

[108] Subramanian Kannan, Mark Roberts, Peter Mayes, Dave Brelsford, and Joseph Skovira. *Workload Management with LoadLeveler*. IBM Press, 2001. 27

[109] Helen D. Karatza and Ralph C. Hilzer. Parallel job scheduling in homogeneous distributed systems. *SIMULATION*, 2003. 19

[110] Katarzyna Keahey and Tim Freeman. Contextualization: Providing One-Click Virtual Clusters. In *ESCIENCE '08: Proceedings of the 2008 Fourth IEEE International Conference on eScience*, pages 301–308, Washington, DC, USA, 2008. IEEE Computer Society. 17

[111] Katarzyna Keahey, Mauricio Tsugawa, Andrea Matsunaga, and Jose Fortes. Sky Computing. *IEEE Internet Computing*, 13(5):43–51, 2009. 18

[112] Attila Kertesz, Peter K. Kacsuk, Ivan Rodero, Francesc Guim, and Julita Corbalan. Meta-Brokering requirements and research directions in state-of-the-art Grid Resource Management. Technical Report TR-0116, Coregrid, 2007. 26

[113] Dalibor Klusacek, Hana Rudova, Ranieri Baraglia, Marco Pasquali, and Gabriele Capannini. Comparison of multi-criteria scheduling techniques. In *Grid Computing Achievements and Prospects*, pages 173–184, 2008. 20

[114] Dalibor Klusáček and Hana Rudová. Alea 2 – Job Scheduling Simulator. In *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques (SIMUTools 2010)*. ICST, 2010. 33

[115] Christopher G. Knight, Sylvia H.E. Knight, Nell Massey, Tolu Aina, Carl Christensen, Dave J. Frame, Jamie A. Kettleborough, Andrew Martin, Stephen Pascoe, Ben Sanderson, David A. Stainforth, and Myles R. Allen. Association of parameter, software and hardware variation with large scale behavior across 57,000 climate models. *Proceedings of the National Academy of Sciences*, 104:12259–12264, 2007. 71

[116] Derrick Kondo, Michela Taufer, Charles L. Brooks III, Henri Casanova, and Andrew A. Chien. Characterizing and Evaluating Desktop Grids: An Empirical Study. In *Parallel and Distributed Processing Symposium, International*, page 26, Los Alamitos, CA, USA, 2004. IEEE Computer Society. 16

[117] E. Laure, C. Gr, S. Fisher, A. Frohner, P. Kunszt, A. Krenek, O. Mulmo, F. Pacini, F. Prelz, J. White, M. Barroso, P. Buncic, R. Byrom, L. Cornwall, M. Craig, A. Di Meglio, A. Djaoui, F. Giacomini, J. Hahkala, F. Hemmer, S. Hicks, A. Edlund, A. Maraschini, R. Middleton, M. Sgaravatto, M. Steenbakkers, J. Walk, and A. Wilson. Programming the grid with gLite. In *Computational Methods in Science and Technology*, page 2006, 2006. 30

[118] Katia Leal, Eduardo Huedo, and Ignacio M. Llorente. A decentralized model for scheduling independent tasks in Federated Grids. *Future Generation Computer Systems*, 25(8):840–852, 2009. 26

[119] Lee, Schartzman, Hardy, and Snavely. Are User Runtime Estimates Inherently Inaccurate? In *10th Workshop on Job Scheduling Strategies for Parallel Processing*, pages 253–263, 2004. 4

[120] Alexander Lenk, Markus Klems, Jens Nimis, Stefan Tai, and Thomas Sandholm. What's inside the Cloud? An architectural map of the Cloud landscape. In *CLOUD '09: Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, volume 0, pages 23–31, Washington, DC, USA, May 2009. IEEE Computer Society. 17

[121] Renaud Lepère, Denis Trystram, and Gerhard J. Woeginger. Approximation Algorithms for Scheduling Malleable Tasks under Precedence Constraints. In *ESA '01: Proceedings of the 9th Annual European Symposium on Algorithms*, pages 146–157, London, UK, 2001. Springer-Verlag. 19

[122] David A. Lifka. The ANL/IBM SP scheduling system. In *In Job Scheduling Strategies for Parallel Processing*, pages 295–303. Springer-Verlag, 1995. 24

[123] Dudy Lim, Yew-Soon Ong, Yaochu Jin, Bernhard Sendhoff, and Bu-Sung Lee. Efficient Hierarchical Parallel Genetic Algorithms using Grid computing. *Future Gener. Comput. Syst.*, 23(4):658–670, 2007. 16

[124] Chin Lu and Sau-Ming Lau. An Adaptive Load Balancing Algorithm for Heterogeneous Distributed Systems with Multiple Task Classes. In *International Conference on Distributed Computing Systems*, pages 629–636, 1996. 82

[125] Gurvan Madec. *NEMO Reference manual, ocean dynamic component: NEMO-OPA*. Number 27. Institut Pierre Simon Laplace (IPSL), 2006. ISSN 1288-1619. 71

[126] Muthucumaru Maheswaran, Shoukat Ali, Howard Jay Siegel, Debra Hensgen, and Richard F. Freund. Dynamic Matching and Scheduling of a Class of Independent Tasks onto Heterogeneous Computing Systems. *Journal of Parallel and Distributed Computing*, 59:107–131, 1999. 38

[127] Satoshi Matsuoka, Hidemoto Nakada, Mitsuhisa Sato, and Satoshi Sekiguchi. Design Issues of Network Enabled Server Systems for the Grid. *Grid Computing — GRID 2000*, pages 59–85, 2000. 30

[128] Hashim Mohamed and Dick Epema. Koala: a co-allocating grid scheduler. *Concurrency and Computation: Practice and Experience*, 20(16):1851–1876, 2008. 5

[129] Ahuva W. Mu'alem and Dror G. Feitelson. Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 12(6):529–543, 2001. 4, 23, 24

[130] Aline P. Nascimento, Cristina Boeres, and Vincent E.F. Rebello. Dynamic self scheduling for parallel applications with task dependencies. In *6th International Workshop on Middleware for Grid Computing - MGC 2008 6th International Workshop on Middleware for Grid Computing - MGC 2008 Proceedings of the 6th International Workshop on Middleware for Grid Computing (MGC 2008)*, 2008. 19

[131] Marco A. S. Netto and Rajkumar Buyya. Rescheduling co-allocation requests based on flexible advance reservations and processor remapping. In *9th IEEE/ACM International Conference on Grid Computing (GRID'08)*, pages 144–151, Tsukuba, Japon, September 2008. 5

[132] Marco A. S. Netto and Rajkumar Buyya. Offer-based scheduling of deadline-constrained bag-of-tasks applications for utility computing systems. In *Proceedings of the 18th International Heterogeneity in Computing Workshop (HCW'09), in conjunction with the 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS'09)*, May 2009. 19

[133] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global arrays: a nonuniform memory access programming model for high-performance computers. *J. Supercomput.*, 10(2):169–189, 1996. 13

[134] Tchimou N'Takpé and Frédéric Suter. Concurrent scheduling of parallel task graphs on multi-clusters using constrained resource allocations. In *10th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC'09)*, Rome, Italy, May 2009. 5

[135] Tchimou N'Takpé, Frédéric Suter, and Henri Casanova. A Comparison of Scheduling Approaches for Mixed-Parallel Applications on Heterogeneous Platforms. In *ISPDC '07: Proceedings of the Sixth International Symposium on Parallel and Distributed Computing*, Washington, DC, USA, July 2007. IEEE Computer Society. 19

[136] Taikan Oki and Yogesh C. Sud. Design of Total Runoff Integrating Pathways (TRIP). Technical Report 2, Earth Integration, 1998. 71

[137] OMG. Corba component model, v4.0, 2010. `http://www.omg.org/technology/documents/formal/components.htm`. 30

[138] Andrei Radulescu and Arjan J. C. van Gemund. A low-cost approach towards mixed task and data parallel scheduling. In *ICPP '02: Proceedings of the 2001 International Conference on Parallel Processing*, pages 69–76, Washington, DC, USA, 2001. IEEE Computer Society. 19

[139] Andrei Radulescu, C. Nicolescu, Arjan J. C. van Gemund, and Pieter Jonker. CPR: Mixed task and data parallel scheduling for distributed systems. In *IEEE International Parallel and Distributed Processing Symposium*, page 39. IEEE Computer Society, 2001. 74, 76

[140] Andrei Radulescu and Arjan J. C. van Gemund. Low-cost mixed task and data parallel scheduling. In *30-th International Conference on Parallel Processing (ICPP)*, pages 69–76, August 2001. 74, 76

[141] Shankar Ramaswamy, Sachin Sapatnekar, and Prithviraj Banerjee. A Framework for Exploiting Data and Functional Parallelism on Distributed Memory Multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 8(11):1098–1116, November 1997. 74

[142] Kavitha Ranganathan and Ian Foster. Decoupling Computation and Data Scheduling in Distributed Data-Intensive Applications. In *HPDC '02: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, page 352, Washington, DC, USA, 2002. IEEE Computer Society. 33

[143] Thomas Rauber and Gudula Runger. Compiler support for task scheduling in hierarchical execution models. *Journal of Systems Architecture*, 45(6-7):483–503, 1999. 74

[144] George F. Riley. The georgia tech network simulator. In *MoMeTools '03: Proceedings of the ACM SIGCOMM workshop on Models, methods and tools for reproducible network research*, pages 5–12, New York, NY, USA, 2003. ACM. 33

[145] Bhaskar Prasad Rimal, Eunmi Choi, and Ian Lumb. A Taxonomy and Survey of Cloud Computing Systems. *Networked Computing and Advanced Information Management, International Conference on*, 0:44–51, 2009. 18

[146] Benny Rochwerger, David Breitgand, Eliezer Levy, Alex Galis, Kenneth Nagin, Ignacio M. Llorente, Ruben Montero, Yaron Wolfsthal, Erik Elmroth, Juan Caceres, Muli Ben-Yehuda, Wolfgan Emmerich, and Fermin Galan. The RESERVOIR Model and Architecture for Open Federated Cloud Computing. *IBM Journal of Research and Development*, 53(4), 2009. 18

[147] Ivan Rodero. *Coordinated Scheduling and Resource Management for Heterogeneous Clusters and Grid Systems*. PhD thesis, Universitat Politecnica de Catalunya, 2009. 26

[148] Ivan Rodero, Francesc Guim, Julita Corbalan, Liana Fong, and S. Masoud Sadjadi. Grid broker selection strategies using aggregated resource information. *Future Generation Computer Systems*, 0:15, August 2009. 5

[149] Jonathan M. Romaine. *Solving the Multidimensional Multiple Knapsack Problem with Packing constraints using Tabu Search*. PhD thesis, Air Force Institute of Technology, 1999. 82

[150] Mitsuhisa Sato, Hidemoto Nakada, Satoshi Sekiguchi, Satoshi Matsuoka, Umpei Nagashima, and Hiromitsu Takagi. Ninf: A network based information library for global world-wide computing infrastructure. In *HPCN Europe*, pages 491–502, 1997. 5, 31

[151] Curt Schimmel. *UNIX systems for modern architectures: symmetric multiprocessing and caching for kernel programmers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994. 12

[152] Uwe Schwiegelshohn, Andrai Tchernykh, and Ramin Yahyapour. Online scheduling in grids. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–10, April 2008. 26

[153] Uwe Schwiegelshohn and Ramin Yahyapour. Analysis of first-come-first-serve parallel job scheduling. In *SODA'98: Procs of the 9th annual ACM-SIAM symposium on Discrete algorithms*, pages 629–638, Philadelphia, PA, USA, 1998. Society for Industrial and Applied Mathematics. 23

[154] K. Seymour, C. Lee, F. Desprez, H. Nakada, and Y. Tanaka. The End-User and Middleware APIs for GridRPC. In *Workshop on Grid Application Programming Interfaces, In conjunction with GGF12*, Brussels, Belgium, September 2004. 30

[155] Edi Shmueli and Dror G. Feitelson. Backfilling with lookahead to optimize the packing of parallel jobs. *Journal of Parallel and Distributed Computing*, 65(9):1090–1107, 2005. 25

[156] Ozan Sonmez, Nezih Yigitbasi, Alexandru Iosup, and Dick Epema. Trace-Based Evaluation of Job Runtime and Queue Wait Time Predictions in Grids. In *International Symposium on High Performance Distributed Computing (HPDC'09)*, June 11-13 2009. 36, 37

[157] Jaspal Subhlok and Gary Vondran. Optimal Use of Mixed Task and Data Parallelism for Pipelined Computations. *Journal of Parallel and Distributed Computing*, 60(3):297–319, 2000. 75

[158] Rajesh Sudarsan and Calvin J. Ribbens. Scheduling resizable parallel application. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS'09)*, 2009. 19

[159] Rajesh Sudarsan and Calvin J. Ribbens. Design and performance of a scheduling framework for resizable parallel applications. *Parallel Computing*, 36:48–64, 2010. 56

[160] Woodruff T. Sullivan III, Dan Werthimer, Stuart Bowyer, Jeff Cobb, David Gedye, and David Anderson. A new major seti project based on project serendip data and 100,000 personal computers. 16

[161] SUN Microsystems. Java rmi, 2010. `http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp`. 30

[162] Atsuko Takefusa, Satoshi Matsuoka, Henri Casanova, and Francine Berman. A Study of Deadline Scheduling for Client-Server Systems on the Computational Grid. In *HPDC '01: Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing*, page 406, Washington, DC, USA, 2001. IEEE Computer Society. 33

[163] Y. Tanimura, K. Seymour, E. Caron, A. Amar, H. Nakada, Y. Tanaka, and F. Desprez. Interoperability Testing for The GridRPC API Specification. In *Open Grid Forum Informational Document*, May 2007. 30

[164] Andrei Tchernykh, Juan Manuel Ramírez, Arutyun Avetisyan, Nikolai Kuzjurin, Dmitri Grushin, and Sergey Zhu. Two Level Job-Scheduling Strategies for a Computational Grid. In *LNCS*, 2006. 25

[165] Haluk Topcuouglu, Salim Hariri, and Min-you Wu. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. *IEEE Trans. Parallel Distrib. Syst.*, 13(3):260–274, 2002. 75

[166] Dan Tsafrir, Yoav Etsion, and Dror G. Feitelson. Backfilling Using System-Generated Predictions Rather than User Runtime Estimates. *IEEE Transactions on Parallel and Distributed Systems*, 18(6):789–803, 2007. 4

[167] Sophie Valcke, Arnaud Caubel, Reiner Vogelsang, and Damien Declat. Oasis 3, User Guide. Technical Report PRISM Report Serie no 2 (5th edition), CERFACS, Toulouse, 2004. 71

[168] Pedro Velho and Arnaud Legrand. Accuracy Study and Improvement of Network Simulation in the Simgrid Framework. In *SIMUTools'09, 2nd International Conference on Simulation Tools and Techniques*, 2009. 34

[169] Constantino Vázquez, Eduardo Huedo, Rubén S. Montero, and Ignacio M. Llorente. Federation of TeraGrid, EGEE and OSG infrastructures through a metascheduler. *Future Generation Computer Systems*, 26(7):979–985, 2010. 5, 26

[170] Lizhe Wang, Gregor von Laszewski, Jay Dayal, and Fugang Wang. Towards Energy Aware Scheduling for Precedence Constrained Parallel Tasks in a Cluster with DVFS. In *10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 368–377. IEEE, 2010. 22

[171] Chuliang Weng and Xinda Lu. Heuristic scheduling for bag-of-tasks applications in combination with qos in the computational grid. *Future Generation Computer Systems*, 21(2):271–280, 2005. 19

[172] Chengzhong Xu and Francis C. Lau. *Load Balancing in Parallel Computers: Theory and Practice*. Kluwer Academic Publishers, Norwell, MA, USA, 1997. 19

[173] Jianhui Yue. Global Backfilling Scheduling in Multiclusters. *Lecture notes in Computer Science*, 3285:232–239, 2004. 37

[174] Yanyong Zhang, Hubertus Franke, Jose Moreira, and Anand Sivasubramaniam. An Integrated Approach to Parallel Scheduling Using Gang-Scheduling, Backfilling, and Migration. *IEEE Trans. Parallel Distrib. Syst.*, 14(3):236–247, 2003. 22

[175] Henan Zhao and Rizos Sakellariou. Scheduling multiple DAGs onto heterogeneous systems. In *20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*, page 14. IEEE Computer Society, April 2006. 5, 74

Abstract:

The work presented in this Thesis is about scheduling applications in computational Grids. We study how to better manage jobs in a grid middleware in order to improve the performance of the platform. Our solutions are designed to work at the middleware layer, thus allowing to keep the underlying architecture unmodified.

First, we propose a reallocation mechanism to dynamically tackle errors that occur during the scheduling. Indeed, it is often necessary to provide a runtime estimation when submitting on a parallel computer so that it can compute a schedule. However, estimations are inherently inaccurate and scheduling decisions are based on incorrect data, and are therefore wrong. The reallocation mechanism we propose tackles this problem by moving waiting jobs between several parallel machines in order to reduce the scheduling errors due to inaccurate runtime estimates.

Our second interest in the Thesis is the study of the scheduling of a climatology application on the Grid. To provide the best possible performances, we modeled the application as a Directed Acyclic Graph (DAG) and then proposed specific scheduling heuristics. To execute the application on the Grid, the middleware uses the knowledge of the application to find the best schedule.

Keywords:

Grid computing, scheduling, reallocation, grid middleware, scientific computing

Résumé :

Les travaux présentés dans cette thèse portent sur l'ordonnancement d'applications au sein d'un environnement de grille de calcul. Nous étudions comment mieux gérer les tâches au sein des intergiciels de grille, ceci dans l'objectif d'améliorer les performances globales de la plateforme. Les solutions que nous proposons se situent dans l'intergiciel, ce qui permet de conserver les architectures sous-jacentes sans les modifier.

Dans un premier temps, nous proposons un mécanisme de réallocation permettant de prendre en compte dynamiquement les erreurs d'ordonnancement commises lors de la soumission de calculs. En effet, lors de la soumission sur une machine parallèle, il est souvent nécessaire de fournir une estimation du temps d'exécution afin que celle-ci puisse effectuer un ordonnancement. Cependant, les estimations ne sont pas précises et les décisions d'ordonnancement sont sans cesse remises en question. Le mécanisme de réallocation proposé permet de prendre en compte ces changements en déplaçant des calculs d'une machine parallèle à une autre.

Le second point auquel nous nous intéressons dans cette thèse est l'ordonnancement d'une application de climatologie sur la grille. Afin de fournir les meilleures performances possibles nous avons modélisé l'application puis proposé des heuristiques spécifiques. Pour exécuter l'application sur une grille de calcul, l'intergiciel utilise ces connaissances sur l'application pour fournir le meilleur ordonnancement possible.

Mots-clés :

Grille de calcul, ordonnancement, reallocation, intergiciel de grille, calcul scientifique