Contribution au Déploiement d'un Intergiciel Distribué et Hiérarchique, Appliqué aux Simulations Cosmologiques

Benjamin Depardon

N° d'ordre : 585 *N° attribué par la bibliothèque : ___ENSL585*

- 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 de Mathématiques et Informatique fondamentale

présentée et soutenue publiquement le 6 octobre 2010 par

M. Benjamin DEPARDON

Contribution to the Deployment of a Distributed and Hierarchical Middleware Applied to Cosmological Simulations

Directeur de thèse :	M.	Eddy	CARON	
Après avis de :	M.	Thierry	PRIOL	
	M.	Vincent	VILLAIN	

Devant la commission d'examen formée de :

M.	Eddy	CARON	Membre
Mme.	Hélène	COURTOIS	Membre
M.	Frédéric	DESPREZ	Membre
M.	Emmanuel	JEANNOT	Membre
M.	Thierry	PRIOL	Membre/Rapporteur
M.	Vincent	VILLAIN	Membre/Rapporteur

À Leïla et Cyril.

Merci !

Ouf ! Quelques années en arrière je ne me voyais pas du tout faire une thèse. Ça me semblait tellement loin, impossible à réaliser. Et voila, vous la tenez entre vos mains, trois ans résumés en ces quelques pages. Une étape de plus de franchie.

J'aimerais tout d'abord dire merci aux membres du jury, Hélène Courtois pour le temps passé à m'expliquer en détail le contenu de tes recherches en cosmologie, Emmanuel Jeannot pour avoir accepté de présider le jury, Thierry Priol et Vincent Villain pour avoir pris le temps de relire ma thèse bien que le contenu des différentes parties ne correspondait pas forcément à vos domaines de recherche, c'est ma faute, j'aime toucher à tout, je ne peux m'empêcher d'essayer différentes approches, d'autant plus si elles sont amusantes.

Je tiens ensuite à remercier mon directeur de thèse, Eddy, pour ces trois ans passés à travailler à tes côtés. Pour m'avoir emmené bosser à l'autre bout du monde à Hawaii ou entre quelques excursions au milieu des poissons on travaillait dans la seule pièce je pense du labo qui ressemblait plus à un débarras qu'à un bureau ; ou encore Las Vegas où les rôles ont été inversés puisque je gardais un oeil sur l'avancée de ta rédaction d'HDR ;-) Je tenais également à te remercier pour m'avoir involontairement "convaincu" de passer mon permis moto suite à un trajet à l'aéroport pour ma première mission LEGO. Je suis arrivé à l'aéroport avec un sourire jusqu'aux oreilles et mal au cou de ne pas avoir réussi à tenir ma tête correctement face au vent. Merci enfin pour ta bonne humeur permanente, même quand tu te couches à 4h du mat' (comme tous les soirs…), et pour m'avoir accordé ta confiance sur tant de points.

Un grand merci également à Fred. Tu n'es pas marqué en tant que directeur de thèse sur ce manuscrit, même si tu le mériterais grandement, je t'ai toujours considéré comme mon deuxième directeur de thèse. Ça a vraiment été du bonheur de travailler avec toi. Je crois que tu a bien plus cru en cette thèse que moi. J'ai tendance à toujours avoir un regard très critique sur ce que je fais, et j'avoue que ça rassurait de t'entendre dire que non, finalement ce je ne faisais pas que de la merde. Merci aussi pour toutes ces discussions devant un tableau malgré le peu de temps que tu avais du fait de tes charges administratives.

Merci également à tous ceux qui ont participé indirectement au bon déroulement de cette thèse, et qui ont fait de la vie au labo soit si intéressante. Cédric pour les courses de chaises dans les couloirs du LIP et les "bastons" à grand coup de rouleaux en carton le soir quand le labo était vide. JS pour sa bonne humeur et pour le temps passé ensemble lors de mon premier IPDPS. Christian et Julien pour leurs trolls lâchés en douce au coin café, et qui animaient bien les discussions. Ghislain pour ses blagues plus nulles les unes que les autres, et les phrases du jour écrites au tableau. Matthieu, Flo et Adrian pour nos séances de sport le midi et surtout l'escalade, rien de tel pour décompresser (enfin… ça me stresse toujours autant d'être au dessus du clou).

Enfin, un grand merci à ma famille et mes amis qui m'ont supportés pendant ces trois ans. Je crois que mon travail a toujours été incompréhensible pour eux, malheureusement ce manuscrit en Anglais ne va pas aider à arranger les choses, mais j'espère qu'au moins la soutenance à clarifié un peu tout ça.

Contents

Introduction

Cosmologists have always had their eyes in the sky. However, even the biggest telescopes cannot unveil all the mysteries of the universe, and sooner or later, they have to put their feet back on the ground, and simulate what observations cannot provide. As research advances, newer and more accurate models appear. They need to be challenged, compared to real observations. Adding precision, or simulating bigger and bigger parts of the universe requires a lot of computing power that no current computer can provide on its own.

So scientists first thought about creating massively parallel computers, known as *supercomputers*, composed of thousands of identical processors interconnected by a dedicated and high speed network. Then appeared loosely coupled, but identical machines, grouped into *clusters*, which was a cheaper solution than supercomputers. Finally, the creation of wide area networks allowed the aggregation of heterogeneous resources distributed around the world, into what is called a *grid*. More recently, the grid evolved, thanks to virtualization techniques, into the *cloud*.

The emergence of *grid computing* has been highly beneficial for many research domains, as it provides a large amount of computing power. Sadly, using such an environment is not as straightforward as it should be.

From Observations to Simulations

People have always been fascinated by the night sky and its million tiny dots of light. Study of the universe goes back in time. Already during Classical Antiquity, the Milky Way and movement of the planets were studied. However, the real breakthroughs in this domain appeared with the creation of new observation instruments. Thus, in 1610 Galileo Galilei, with the help of his new terrestrial telescope, discovered three of Jupiter's four largest satellites. Thereon, the instruments became more and more accurate. In 1845, Lord Rosse constructed a new telescope and was able to distinguish between elliptical and spiral nebulae. However, even if these tools become more powerful, and bring light upon unanswered questions, they also show the limitation of the current cosmological models.

In 1934, Fritz Zwicky observed that the orbital velocities of seven galaxies in the Coma cluster did not follow the theoretical predictions. The galaxies appeared 400 times "heavier" than what was observable. He postulated the existence of an invisible matter, which will be called *Dark Matter*. This theory is somewhat forgotten until the 1970s, when new observations made by Vera Rubin endorse the hypothesis of Fritz Zwicky. This theory is nowadays extensively studied, and large observational surveys are conducted to try to determine the distribution of dark matter.

Dark matter has an important role in cosmological models, has it explains the manner large structures and galaxies have been formed in the universe. However, as it cannot directly be observed, physicists rely either on statistical observational data, or simulations. Cosmology based on simulations is a quite recent field of research. A cosmological simulation amounts, neither more nor less, to evolving "all" the particles of the universe, from the Big Bang, up until now. This involves solving complex equations, that no human can do in a reasonable time. Thus, simulations are conducted on computers, either large parallel computers, or on a grid of computers. For example, the Horizon Project [10] has executed a 13.7 Gigayear long simulation, during which the evolution of 4096^3 dark matter particles has been studied. This required the concurrent utilization of 6144 processors of the BULL supercomputer of the CEA (French atomic energy commission) supercomputing center.

Cosmological simulations is only one among many examples of the usage of computing resources to solve large scientific problems. Grid computing is nowadays widely used in heterogeneous domains such as biology, cosmology, fluid mechanics, high energy physics, or climatology.

Dealing with Increasing Computational Needs

Despite the rapid increase of computational power available within each computer, a single machine is soon no longer sufficient to run complex scientific applications. Coping with this problem has been the center of interest of computer scientists for many years now. New technologies have been developed, and computing resources became more accessible. The first step has been to gather several processors around a central and unique memory shared by all processing units. The next step has been to group independent computers into *clusters*, each processor having its own memory. Finally, arose *Grid Computing*, which primary goal is to aggregate resources distributed around the world (belonging to public or private institutions) through wide area networks, into a virtual entity called the *grid*, which could easily, and hopefully transparently provide an "unlimited" computing power to end-users.

However, this attractive enormous amount of computing power available in grids is only the frontage of a complex infrastructure. Reality is still far from this idealistic view expressed by Foster and Kesselman [95], where a computational grid would be as easy to use as the electrical power grid. Several barriers need to be removed before achieving this transparency. Problems are present at several levels. Among them, we find the problem of accessing resources through a *middleware*, whose goal is to grant or deny access to users based on security policies, and to select resources that would best fit the users' requirements. Scheduling, whose goal is to efficiently organize computing jobs to avoid unfairness between users, or ensure an efficient utilization of the platform, it also ensures that dependencies between jobs are respected. Data management, whose goal is to store, replicate and move data on the platform according to the needs, so as to provide good performances on the computations. Fault-tolerance, whose goal is to provide recovery mechanisms and ensure service availability in case of crashes or failures, which are, statistically, very likely to occur on very large and dynamic platforms such as grids. Scalability, whose goal is to ensure that the growing number of users, jobs, and resources will not impact the efficiency of the applications and the response time expected by the users.

Apart from the number of involved machines, and the fact that they are distributed at a very large scale, the above mentioned problems are made even more complicated, and even sometimes intractable, by one of the main characteristics of the grid, *heterogeneity*, in terms of hardware (processors' computing power, available memory and storage space, or network availability and performance), and in terms of software (operating system, available libraries, or communication protocols).

All in all, these problems are related with the efficient management of both the platform and the jobs. If for expert users using a grid can already be bothersome, it becomes a real challenge for users from other domains such as biology, or physics. Seen from the users' point of view, all the complexity should be hidden, and only a few basic commands should be sufficient to submit jobs. From the administrators' point of view, this requires an efficient deployment of the middleware in charge of the resources, its constant adaptation to modifications, or errors occurring in the system, efficient scheduling techniques, and finally an easy to use front end for the end-users.

Dissertation Organization

This dissertation is organized into four independent (but yet related) parts. They follow a common goal which is to provide non-expert users an efficient and transparent way to access heterogeneous and distributed computational resources. The first three parts are divided into two chapters each: a chapter presenting the necessary background and related work, and a chapter explaining the problem we are tackling along with the proposed solutions. The last part being on a more practical side, is only composed of one chapter.

The first part of this dissertation deals with the problem of accessing resources. In Chapter 1, we introduce grid computing, along with the means of accessing its computing power, *i.e.,* middleware. Cosmological simulations, just like any other scientific computation, need to rely on an effective and transparent mean of accessing resources. The most scalable middleware rely on a hierarchical structure, Chapter 2 presents our solution to automatically determine the shape of a hierarchical middleware, such that it provides the highest possible throughput of executed jobs, *i.e.,* the maximum number of jobs finished per time unit, when several applications are executed concurrently. As many different kinds of computing platforms exist, we consider three cases which encompass all possibilities, namely: fully homogeneous platforms, communication homogeneous/computation heterogeneous platforms, and fully heterogeneous platforms. We provide heuristics based on linear programs, or on a genetic algorithm, and compare the theoretical results with experiments conducted on a real platform.

In the second part, we address the problem of identifying groups of well connected nodes in a dynamic and error prone network. The goal is to provide quality of service on the communication latency a request sent to a middleware would undergo. We present in Chapter 3 general techniques for clustering a graph, either in a centralized or in a distributed manner. We also present self-stabilization, which is a paradigm for designing fault-tolerance and auto-adaptive algorithms. In Chapter 4, we tackle the problem of grouping nodes of a weighted graph into clusters with bounded diameters. This clustering ensures that the time to communicate between any two nodes within a cluster is bounded. Our solution is totally distributed, and self-stabilizing under an unfair daemon.

The third part is dedicated to the scheduling of independent tasks on a platform composed of several clusters. Chapter 5 gives background and related work on scheduling. As post-processing of cosmological simulations can be seen as bags-of-tasks, which require to be efficiently scheduled on the grid. In Chapter 6, we present our solution to schedule independent tasks on clusters which utilization is limited, *i.e.,* the computing power a user has access to on a period of time may not be the total available computing power. We deal with this problem in two cases: when the tasks arrive all at the same time, and when release dates are considered.

Finally, the last part of this dissertation presents a more practical aspect of our contribution, namely the execution of cosmological simulations on a computational grid. We present a complete infrastructure for easily accessing a grid, and submitting workflows of cosmological simulations. It relies on a web site on the user side, that communicates with a middleware which is in charge of handling the workflows. The communications are also being taken care of when firewalls prevent direct communications between nodes.

Part I

Deploying Trees

Chapter 1

Preliminaries on Grid Computing

In this chapter, we give information and related work required as a background to understand the first part of this dissertation on the *deployment of a Grid-RPC middleware.* We first present, in the next section, the concept of *Grid Computing*: we recall the definition of a *Grid*, and give examples of several projects providing computing infrastructures. In Section 1.2, we give an overview of the existing *grid middleware*, *i.e.,* means of accessing grids computing power. We particularly focus on one simple and yet efficient approach to build grid middleware: the *Grid-RPC* approach, and we present the architecture of one of its implementation: the Diet middleware. Finally, we present in Section 1.3, related work on the problems we are addressing in the next chapter, as well as optimization techniques we make use of.

1.1 Grid Computing

Molecular biology, astrophysics, high energy physics, those are only a few examples among the numerous research fields that have needs for tremendous computing power, in order to execute simulations, or analyze data. Increasing the computing power of the machines to deal with this endlessly increasing needs has its limits. The natural evolution was to divide the work among several processing units. *Parallelism* was first introduced with monolithic parallel machines, but the arrival of high-speed networks, and especially *Wide Area Network* (WAN) made possible the concept of *clusters* of machines, which were further extended to large scale distributed platforms, leading to a new field in computer science, *grid computing.*

The first definition of a *grid* has been given by Foster and Kesselman in [95]. A grid is a distributed platform which is the aggregation of heterogeneous resources. They do an analogy with the electrical power grid. The computing power provided by a grid should be transparently made available from everywhere, and for everyone. The ultimate purpose is to provide to scientific communities, governments and industries an unlimited computing power, in a transparent manner. This raised lots of research challenges, due to the complexity of the infrastructure. Heterogeneity is present at all levels, from the hardware (computing power, available memory, interconnection network), to the software (operating system, available libraries and software), via the administration policies.

From this definition, several kinds of architectures were born. One of the most commonly used architecture, referred to as *remote cluster computing*, is composed of the aggregation of many networked loosely coupled computers, usually those computers are grouped into *clusters* of homogeneous and well connected machines. These infrastructures are often dedicated to scientific or industrial needs, and thus provide large amount of computing resources, and a quite good stability. Users from multiple administrative domains can collaborate and share resources by creating a *Virtual Organization* (VO): a VO grants access to a subset of available machines, to a group of users.

We will now present several grid projects, as well as examples of applications that require huge amounts of computing power, and which can make the most of distributed infrastructures.

1.1.1 Examples of Grid Infrastructures

In this section, we choose to present six national or international grid infrastructures, which are representative of existing platforms. We divide them into two categories: research and production grids. Research grids aim at providing a platform to computer scientists, so that they can compare their theoretical research with real executions in a controlled environment. Whereas production grids are stable environment aiming at executing applications for other research fields.

Research grids

ALADDIN-Grid'5000 [51] is a French project, supported by the French ministry of research, regional councils, INRIA, CNRS and universities, whose goal is to provide an experimental testbed for research on Grid computing since 2003. It provides a nation wide platform, distributed on 9 sites, 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 [21]. 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. Grid'5000 has also already been interconnected with several other grids such as DAS-3 [4] in the Netherlands, and NAREGI [16] in Japan, and future new sites will be connected outside France such as in Brazil and Luxembourg.

FutureGrid [7] is a recent American project, started in October 2009, mimicking Grid'5000 infrastructure. FutureGrid's 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, . . .

OneLab [17] 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** [20]. 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. P2P technologies allow robust, fault tolerant tools for content distribution. They rely on totally decentralized systems in which all basic entities, called *peers*, are equivalent and perform the same task. Though initially outside the scope of grid computing, P2P has progressively gained a major place in grid researches. This convergence between P2P systems and grid computing has been highlighted in [94].

Production grids

EGEE (Enabling Grids for E-sciencE) [5] is a project started in 2004 supported by the *European Commission*. It aims at providing researchers in academia or business, access to a production level Grid Infrastructure. EGEE 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 EGEE platform. Since the end of April 2010, the EGEE project is no longer active. The distributed computing infrastructure is now supported by the **European Grid Infrastructure, EGI** [6].

TeraGrid [23] is an American project supported by the NFS. 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 quite heterogeneous, as one can find clusters of PCs, vectorial machines, parallel SMP machines or even super calculators. On the whole, there are more than a petaflop of computing capability and more than 30 petabytes of online and archival data storage available to industrials and scientists.

World Community Grid [25] is a project initiated and supported by IBM. This system, contrary to previously presented projects, does not rely on federation of dedicated resources, but on federation of individual computers that individuals are willing to share for research. This type of platform is quite different from dedicated computing platforms, as there is no dedicated network, and computing resources cannot be reliable as usually computations only take place when the computer goes to sleep, *i.e.,* when the user does not use it. The researches done on the World Community Grid are mainly medical researches. On its current state, there are about 506,000 users who provide a total of 1,505,860 machines. This platform belongs to another branch of grid computing: *desktop computing*, also known as *volunteer computing*. It relies on the principle that personal computers (PCs) sold to the public are nowadays quite powerful, but rarely used to their full capabilities: most of the time, the owners leave their computers into an idle state. Thus, arose in the 1990s the idea to use *cycle stealing* on those computers to make useful research, with two main projects: Great Internet Mersenne Prime Search in 1996, followed by Distributed.net in 1997. The idea is to use the idle time of PCs connected to the Internet, to run research software. As the performance of PCs and the number of users keeps on increasing, the available computing power allows to solve huge problems.

Using such above mentioned platforms to solve large problems ranging from numerical simulations to life science is nowadays a common practice [45, 95]. We will now give examples of such applications ported on a grid.

1.1.2 What Kind of Applications do we Find in Grid Computing?

Grid infrastructures can be used to solve many types of problems. An application can be *computation intensive* if it requires lots of computations, or *I/O intensive* if it requires lots of Input/Output operations; it can also be *sequential* if it needs to be executed on a single processor, or on the contrary *parallel* if it runs on several processors. Thus, Grid Computing can in fact encompasses many a research field, and many applications (or services) are available. We now present some of the commonly found applications on grid infrastructures.

Lots of simulations such as acoustics and electromagnetics propagation or fluid dynamics can be represented as a set of *linear equations* stored in very large, but sparse, matrices. Several *sparse matrix solvers* [30, 74] can also be executed on grid infrastructures. When not relying on linear equations, problems can also be represented by *differential equations*, which require a different solving approach. A widely used method is *adaptive mesh refinement* (AMR), such as in universe formation simulations [110, 131, 160] (we study such cosmological simulations in Part IV of this dissertation), or climate predictions [85].

Bioinformatics researches use different approaches in order to discover for example new proteins. Some rely on *docking* techniques [82], or on *comparison of DNA sequences* with large databases by using BLAST [29] (Basic Local Alignment Search Tool). Many researches in bioinformatics are realized thanks to the World Community Grid, such as discovering dengue drugs, help find a cure to muscular dystrophy, cancers or even AIDS.

High energy physics is one of the most data and computation consuming research field. The LHC (Large Hadron Collider) [13] is the world's largest and highest-energy particle accelerator. At full operation intensity, the LHC can produce roughly 15 Petabytes of data annually. The data analysis is then realized thanks to the Worldwide LHC Computing Grid project (WLCG) [24] which federates more than 100,000 processors from over 130 sites in 34 countries.

Finally, scientific computing often relies on matrices manipulations. Several linear algebra libraries are thus available either to be integrated directly within an application, or to be remotely called. Among those, we can cite BLAS [84], LAPACK [35] and ScaLAPACK [49], or the SL3 library [153] from SUN. These libraries provide basic linear algebra subroutines, matrix factorizations, eigenvalues computations...

All these techniques and domains of applications are of course only a small subset of what can be found in grid computing.

1.2 Accessing Grid Computing Power

1.2.1 Grid Middleware

Hiding the complexity and heterogeneity of a grid, and providing a transparent access to resources and services is the aim of software tools called *middleware.* A middleware is a layer between the end-user software, and the services and resources of the grid. Its main goals are to provide data management, service execution, monitoring, and security. Two of the most important grid middleware are gLite [121], which is part of the EGEE project, and Globus Toolkit [93]. Both are toolkits aiming at providing a framework for easing the building of grid applications. We can also cite BOINC [33], which is specifically targeted for volunteer computing.

The Grid-RPC Approach

Among grid middleware, a simple, yet efficient approach to provide transparent and productive 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, *i.e.,* it can be a local object on the same machine, or a distant one. In this latter case the communication complexity is hidden. Many RPC implementations exists. Among those, CORBA [134] and Java RMI [152] are the most used.

Several grid middleware [57] are available to tackle the problem of finding services available on distributed resources, choosing a suitable server, then executing the requests, and managing the data. Several environments, called *Network Enabled Servers* (NES) environments [125], have been proposed. Most of them share in common a three main components design: *clients* which are applications that use the NES infrastructure, *agents* which are in charge of handling the clients' requests (scheduling them) and of finding suitable servers, and finally *computational servers* which provide computational power to solve the requests. Some of the middleware only rely on basic hierarchies of elements, a star graph, such as NetSolve/GridSolve [63, 170], Ninf-G [154], OmniRPC [145] and SmartGridRPC [53]. Others, in order to divide the load at the agents level, can have a more complex hierarchy shape such as WebCom-G [129] and DIET [59].

RPC has been specialized in the context of grid computing and gave birth to the *Grid-RPC* [146], thanks to the Open Grid Forum [18]. The Grid-RPC working group from the Open Grid Forum defined a standard Grid-RPC API [157], which allows clients to write their applications regardless of the underlying middleware. Currently, only five NES environments implement this standard API: GridSolve, Ninf-G, OmniRPC, SmartGridRPC, and DIET.

The DIET Middleware

As this dissertation mainly focuses on the DIET middleware, we now present its architecture.

DIET is based on the client-agent-server model. The particularity of DIET is that it allows a hierarchy of agents to distribute the scheduling load among the agents. A DIET hierarchy, as presented in Figure 1.1, starts with a *Master Agent* (MA). This is the entry point, every request has to flow through the MA. A client contacts the MA via the CORBA naming service. The MA can then rely on a tree of *Local Agents* to forward the requests down the tree until they reach the servers. An agent has essentially two roles. First it forwards 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, ...) Finally, 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, this list is exposed to the parent of the SED in the DIET hierarchy. A SED also provides performance prediction metrics which are sent along with the reply whenever a request arrives. These metrics are the building blocks for scheduling policies.

As the MA is the only entry point of the hierarchy, it could become a bottleneck. Thus, to cope with this problem, several DIET hierarchies can be deployed alongside, and interconnected using CORBA connections.

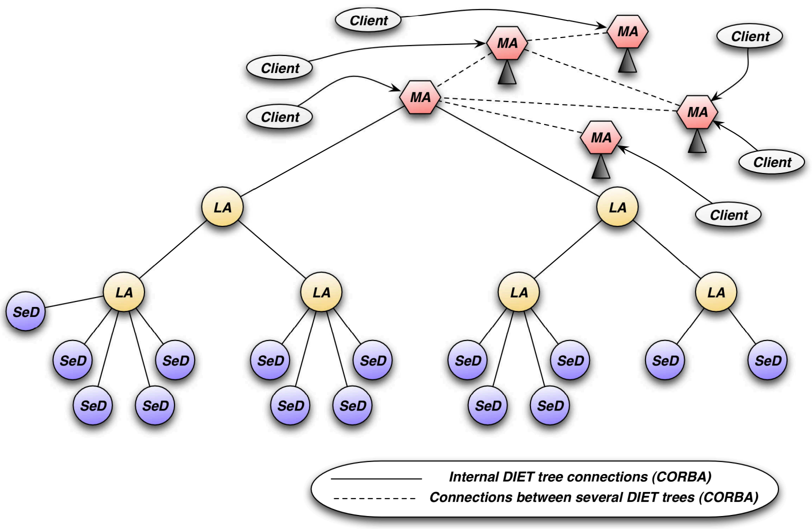

Client
Client
Client
Client
Client
Client
MA
MA
MA
MA
MA
LA
LA
LA
LA
LA
LA
SeD
Internal DIET tree connections (CORBA)
Connections between several DIET trees (CORBA)

Figure 1.1: DIET middleware architecture.

1.2.2 Deployment Software

Another concern is how an element is "really" deployed: how the binaries and libraries are sent to the relevant machines, and how they are remotely executed? This problem appears whenever an application needs to be remotely executed: how is it *deployed*? We can discern three kinds of deployments. The lowest level one consists in deploying *system images*, *i.e.,* installing a new operating system along with relevant software on a machine. This process requires to be able to distantly reboot the node and install whole system image on it. Kadeploy [12] is an example of software to deploy system images on grid environments. At an intermediary level, we find the deployment of *virtual machines*, which basically consists in deploying a system on top of another system. Several virtual machines can then be hosted by a single physical machine. This kind of deployment offers different levels of virtualization, as it can either just virtualize the system without hiding the hardware, or it can emulate another kind of hardware. We can cite Xen [38] and QEMU [43] as examples. Finally, the last level of deployment is *software deployment*. It consists in installing, configuring and executing a piece of software on resources, on top of an already existing operating system.

Deployment is particularly relevant for Grid-RPC middleware, as the deployment needs to be coordinated (servers need to be launched once the agents are ready), and many elements need to be deployed (an agent and/or server per machine). Moreover, some deployment software provide autonomic management, which can be of great help when managing a platform over a long period. Several solutions exists. They range from application specific to totally generic software. ADAGE [119, 120] is a generic deployment software. It relies on modules specific for each software deployment. Its design decouples the description of the application from the description of the platform, and allows specific planning algorithms to be plugged-in. ADAGE is targeted towards static deployment (*i.e.,* "one-shot" deployment with no further modifications), it can however be used in conjunction with CoRDAGe [69] to add basic dynamic adaptations capabilities. ADEM [108] is a generic deployment software, relying on Globus Toolkit. It aims at automatically installing and executing applications on the Open Science Grid (OSG), and can transparently retrieve platform description and information. DeployWare [91, 92] is also a generic deployment software, it relies on the Fractal [137] component model. Mapping between the components and the machines has to be provided, as no automatic planning capabilities are offered. GoDiet [58] is a tool specifically designed to deploy the Diet middleware. An XML file containing the whole deployment has to be provided. TUNe [54] is a generic autonomic deployment software. Like DeployWare, it also relies on the Fractal component model. TUNe targets autonomic management of software, *i.e.,* dynamic adaptation depending on external events. Sekitei [114] is not exactly a "deployment software", as it only provides an artificial intelligence algorithm to solve the component placement problem. It is meant to be used as a mapping component for other deployment software. Finally, Weevil [162, 163] aims at automating experiment processes on distributed systems. It allows application deployment and workload generation for the experiments. However, it does not provide automatic mapping.

Apart from the installation and the execution of the software itself, which has been a well studied field, another important point is the *planning* of the deployment, *i.e.,* the mapping between the software elements, and the computational resources. Whereas deployment software can cope with the installation and execution part, very few propose intelligent planning techniques. Thus the need for proposing planning algorithms.

1.3 Concerning Deployment

In this section, we present some related work on middleware modelization and evaluation, as well as two optimization techniques used in our approach to solve the problem of deploying a Grid-RPC middleware.

1.3.1 Middleware Models and Evaluations

The literature does not provide much papers on the modelization and evaluation of distributed middleware systems. In [155], Tanaka *et al.* present a performance evaluation of Ninf-G. However, no theoretical model is given. In [68], P.K. Chouhan presents a model for hierarchical middleware, and algorithms to deploy a hierarchy of schedulers on cluster and grid environments. She also compares the models with the Diet middleware. She models the maximum throughput of the middleware when it works in steady-state, meaning that only the period when the middleware is fully loaded is taken into account, and not the periods when the workload initially increases, or decreases in the end. However, a severe limitation in this latter work is that only one kind of service could be deployed in the hierarchy. Such a constraint is of course not desirable, as nowadays many applications rely on workflows of different services. Hence, the need to extend the previous models and algorithms to cope with hierarchies supporting several services.

Several works have also been conducted on CORBA-based systems. Works on throughput, latency and scalability of CORBA have been conducted [101, 102], or even on the ORB architecture [27]. Several CORBA benchmarking frameworks have been proposed [55], and some web sites also propose benchmarks results, see for example [142]. These benchmarks provide low level metrics on CORBA implementations, such as consumed memory for each data type, CPU usage for each method call. Though accurate, these metrics do not fit in our study, which aims at obtaining a model of the behavior of Diet at a user level, and not at the methods execution level.

1.3.2 Tools Used in our Approach

We now introduce two techniques used in the next chapter to solve the deployment planning problems. The two methods show quite orthogonal ways of thinking. While *linear programming* aims at obtaining a solution to an optimization problem through exact resolution of a system of constraints, *genetic algorithm* try to find a good solution through random searches and improvements.

Linear Programming

Linear Programming (LP) is a method for the optimization of a linear function (the objective function), subject to several constraints which can be represented as linear equalities or inequalities. More formally, a linear programming problem can be stated in the following form:

$$\begin{array}{l}\textbf{Minimize (or maximize) } c^\top x \\ \textbf{Subject to } Ax \leq b\end{array}$$

Where x is the vector containing the unknown factors (the variables), c and b are vectors containing constant coefficients, and A is a matrix of known coefficients. When all variables can accept a range of floating values, the problem can be solved in polynomial time using the *ellipsoid* or *interior point* algorithms. However, if the unknown variables are all required to be integers, we then talk about *Integer Linear Programming* (ILP), and the optimization problem becomes NP-hard.

Such a technique is widely used, for example for scheduling problems, in operations research, or in our case deployment planning. There exists several software to solve LP problems, such as GNU Linear Programming Kit (GLPK) [8], ILOG CPLEX [11], or lpsolve [14].

Linear programming is used in the next chapter, in Sections 2.3 and 2.4 to design planning algorithms for hierarchical middleware on homogeneous platforms, and communication homogeneous / computation heterogeneous platforms.

Genetic Algorithm

Genetic Algorithms [126] are part of what are called *meta-heuristics* such as *tabu search*, *hill climbing*, or *simulated annealing*. More precisely genetic algorithms are *evolutionary heuristics*. Genetic algorithms were invented by John Holland in the 1970s [107], and are inspired by Darwin's theory about evolution. The idea is to have a *population* of abstract representations (called chromosomes or the genotype of the genome) of candidate solutions (called individuals) to an optimization problem evolve towards better solutions. Starting from a random initial population, each step consists in stochastically selecting individuals from the current population (based on their *fitness*, *i.e.,* their score towards the optimization goal), modifying them by either combining them (using techniques similar to natural *crossovers*) or *mutating* them (randomly modifying the chromosomes), before injecting them into a new population, which in turn is used for the next iteration of the algorithm. Usually, this process is stopped after a given number of iterations, or once a particular fitness is attained.

There exists quite a lot of parallel distributed evolutionary algorithms and local search frameworks, such as DREAM [36], MAFRA [116], MALLBA [28], and ParadisEO [56].

A genetic algorithm is used in the next chapter, in Section 2.5 for designing a planning algorithm for hierarchical middleware on heterogeneous platform.

Chapter 2

Middleware Deployment

As already mentioned in Chapter 1, “the word Grid is used by analogy with the electric power grid, which provides pervasive access to electricity and has had a dramatic impact on human capabilities and society,” (I. Foster in [95]). Though this may have become more or less true for the end-users of the Grid (apart from the fact that I still do not know how to plug my programs as easily as I plug my coffee machine), administrators or service providers still have to deploy the infrastructure, the middleware and the services. End-users can rely on high-level tools (web pages, integrated environments, *etc.*) to submit their jobs, but what about the underlying middleware that has to schedule them? How many machines are necessary to support the load incurred by the clients? Has the middleware to be centralized or distributed in order to cope with the scheduling load?

In this chapter[1], we deal with the problem of deploying a hierarchical grid middleware. We do not address the “practical” problem of how to deploy and execute the binaries, this has been extensively studied, and several deployment software are freely available (see previous chapter). Instead, we concentrate on determining what is the *best* mapping of the middleware on the available resources to fulfill users' requirements. This problem is twofold: we first need to model the behavior of the middleware, then, as we are dealing with hierarchical middleware, we need to determine jointly the shape of the hierarchy, and the mapping of this hierarchy on the nodes. Platforms can also be of different “flavors”: homogeneous or heterogeneous. All in all, four problems are addressed in this chapter:

1. **Modelization.** We provide a model for hierarchical Grid-RPC middleware. It is an extension of the one presented in [68], in that it takes into account several services.
2. **Homogeneous platform.** This is the simplest platform we can deal with: all nodes have the same characteristics, and are interconnected with links which also have the same characteristics. This model reflects the common case of *cluster computing*.
3. **Computation heterogeneous platform.** Going a step further in this model, we consider that nodes can have different characteristics, even though the interconnection links remain homogeneous. This reflects the case where several machines share the same network: for example *clusters* that are linked altogether, or company *desktop computers*.
4. **Heterogeneous platform.** The last step is of course the case where every machine, and every link can have its own characteristics. This is the typical *grid computing* case.

The rest of this chapter is organized as follows. In Section 2.1, we give the assumptions made on both the platform and the middleware. Then, in Section 2.2 we give a generic model for predicting the performance of a hierarchical middleware deployed with several services. We tackle the problems of deploying the middleware on three different types of platforms in Sections 2.3 (homogeneous platform), 2.4 (computation heterogeneous platform), and 2.5 (heterogeneous platform). For each of those, we present an algorithm and experimental results. Finally, we present parameters influencing the middleware performance in Section 2.6.

1. The work presented in this chapter has been published in international conferences [CDD10a] and [CDD10b].

2.1 Model Assumptions

2.1.1 Request Definition

Clients use a 2-phases process to interact with a deployed hierarchy. They submit a scheduling request to a Master Agent to find a suitable server in the hierarchy (scheduling phase), and then submit a service request (job) directly to the server (service phase). A completed request is one that has completed both the scheduling and service request phases and for which a response has been returned to the client. The throughput of a service is the number of completed requests per time unit the system can offer. We consider that a set $\mathcal{R}$ of services has to be available in the hierarchy. And that for each service $i \in \mathcal{R}$, the clients aim at attaining a throughput ρ_i^* of completed requests per second.

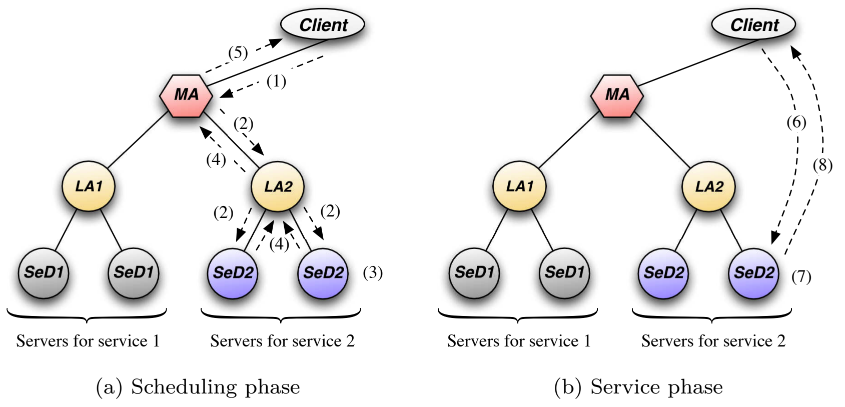


Figure 2.1: Scheduling and service phases.

Figure 2.1 presents the scheduling and service phases. The example shows the following steps:

1. the client sends a request of type i ($i = 2$ in Figure 2.1);
2. the request is forwarded down the hierarchy, but only to the sub-hierarchy that knows service i;
3. the SeDs perform performance predictions, and generate their response;
4. responses are forwarded up the hierarchy, and "sorted", *i.e.,* the best choice is selected at each agent level;
5. the scheduling response is sent back to the client (the response contains a reference to the selected server);
6. the client sends a service request directly to the selected server;
7. the server runs the application and generates the results;
8. the results are directly sent back to the client.

2.1.2 Resource Architecture

We consider the problem of deploying a middleware on a fully connected platform $G = (V, E, W, B)$: V is the set of nodes, E is the set of edges, $w_j \in W$ is the processing power in $Mflops/s$ of node j, and the link between nodes j and j' has a bandwidth $B_{j,j'} \in B$ in $Mbit/s$. We do not take into account contentions in the network. We also denote by $\mathcal{N} = |V|$ the number of nodes.

Remark. *We consider that whatever the node they are running on, services and agents require a constant amount of computation,* i.e., *we consider the case of* uniform machines *where the execution time of an application on a processor is equal to a constant only depending on the machine, multiplied by a constant only depending on the application.*

2.1.3 Deployment Assumptions

We consider that at the time of deployment we do not know the clients' locations, or the characteristics of the clients' resources. Thus, clients are not considered in the deployment process and, in particular, we assume that the set of computational resources used by clients is disjoint from V, and that we will not consider client/servers direct communications in our model (phases 6 and 8 in Figure 2.1). This corresponds to the case where data required for the services is already present in the platform, and does not need to be sent by the clients. A valid deployment will always include at least the root-level agent and one server per service $i \in \mathcal{R}$. Each node $v \in V$ can be assigned either as a server for any kind of service $i \in \mathcal{R}$, or as an agent, or left idle. Thus with $|\mathcal{A}|$ agents, $|\mathcal{S}|$ servers, and $|V|$ total resources, $|\mathcal{A}| + |\mathcal{S}| \leq \mathcal{N}$.

2.1.4 Objective

We consider that we work in steady-state. The platform is loaded to its maximum. Thus, we do not take into account phases when only a few clients are submitting requests to the system, but only the phase where the clients submit as many requests as the middleware is able to cope with.

As we have multiple services in the hierarchy, our goal cannot be to maximize the global throughput of completed requests regardless of the kind of services, this would lead to favor services requiring only small amounts of computing power to schedule and to solve them, and with few communications. Hence, our goal is to obtain, for each service $i \in \mathcal{R}$, a throughput ρ_i such that all services receive almost the same obtained throughput to requested throughput ratio: $\frac{\rho_i}{\rho_i^*}$, of course we try to maximize this ratio, while having as few agents in the hierarchy as possible, so as not to use more resources than necessary.

2.2 Servers and Agents Models

2.2.1 "Global" Throughput

For each service $i \in \mathcal{R}$, we define ρ_{sched_i} to be the scheduling throughput for requests of type i offered by the platform, *i.e.,* the rate at which requests of type i are processed by the scheduling phase. We define as well ρ_{serv_i} to be the service throughput.

Lemma 2.1. *The completed request throughput ρ_i of type i of a deployment is given by the minimum of the scheduling and the service request throughput ρ_{sched_i} and ρ_{serv_i}.*

$$\rho_i = \min\left\{\rho_{sched_i}, \rho_{serv_i}\right\}$$

Proof. A completed request has, by definition, completed both the scheduling and the service request phases, whatever the kind of request $i \in \mathcal{R}$.

Case 1: $\rho_{sched_i} \geq \rho_{serv_i}$. In this case, requests are sent to the servers at least as fast as they can be processed by the servers, so the overall rate is limited by ρ_{serv_i}.

Case 2: $\rho_{sched_i} < \rho_{serv_i}$. In this case, the servers process the requests faster than they arrive. The overall throughput is thus limited by ρ_{sched_i}. ■

Lemma 2.2. *The service request throughput ρ_{serv_i} for service i increases as the number of servers included in a deployment and allocated to service i increases.*

2.2.2 Hierarchy Elements Model

We now describe the model of each element of the hierarchy. We consider that a request of type i is sent down a branch of the hierarchy, if and only if service i is present in this branch, *i.e.,* if at least a server of type i is present in this branch of the hierarchy. Thus a server of type i will never receive a request of type $i' \neq i$. Agents won't receive a request i if no server of type i is present in its underlying hierarchy, nor will it receive any reply for such a type of request. This is the model used by DIET.

Server model

We define the following variables for the servers. w_{pre_i} is the amount of computation in $MFlops$ needed by a server of type i to predict its own performance when it receives a request of type i from its parent. Note that a server of type i will never have to predict its performance for a request of type $i' \neq i$ as it will never receive such a request. w_{app_i} is the amount of computation in $MFlops$ needed by a server to execute a service. m_{req_i} is the size in $Mbit$ of the messages forwarded down the agent hierarchy for a scheduling request, and m_{resp_i} the size of the messages replied by the servers and sent back up the hierarchy. Since we assume that only the best server is selected at each level of the hierarchy, the size of the reply messages does not change as they move up the tree. Figure 2.2 presents the server model.

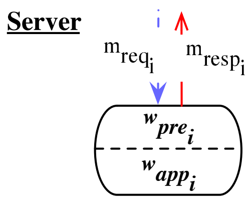


Figure 2.2: Server model parameters.

Server computation model. We suppose that a deployment with a set of servers $\mathcal{S}_i$ completes N_i requests of type i in a given time frame. $\mathcal{S}_i$ being the set of nodes capable of computing requests of type i. Then, each server $j \in \mathcal{S}_i$ will complete N_i^j such that:

$$\sum_{j \in \mathcal{S}_i} N_i^j = N_i \tag{2.1}$$

On average, each server j has to do a prediction for N_i requests, and complete N_i^j service requests during the time frame. Let $T_{comp_i}^{server_j}$ be the time taken by the server $j \in \mathcal{S}_i$ to compute N_i^j requests and predict N_i requests. We have the following equation:

$$T_{comp_i}^{server_j} = \frac{w_{pre_i}.N_i + w_{app_i}.N_i^j}{w_j} \tag{2.2}$$

Now let's consider a time step T during which N_i requests are completed. On this time step, we have:

$$\forall i \in \mathcal{R},\ \forall j \in \mathcal{S}_i,\ T = \frac{w_{pre_i}.N_i + w_{app_i}.N_i^j}{w_j} \tag{2.3}$$

From (2.3), we can deduce the number of requests computed by each server $j \in \mathcal{S}_i$ for all type of requests $i \in \mathcal{R}$:

$$N_i^j = \frac{T.w_j - w_{pre_i}.N_i}{w_{app_i}} \tag{2.4}$$

From equations (2.1) and (2.4), we can rewrite the time taken by the $\mathcal{S}_i$ servers to process N_i requests of type i:

$$\begin{aligned}
\sum_{j \in \mathcal{S}_i} N_i^j &= \sum_{j \in \mathcal{S}_i} \frac{T.w_j - w_{pre_i}.N_i}{w_{app_i}} \\
N_i &= T \times \frac{\sum_{j \in \mathcal{S}_i} w_j}{w_{app_i}\left(1 + \sum_{j \in \mathcal{S}_i} \frac{w_{pre_i}}{w_{app_i}}\right)} \\
N_i &= T \times \frac{\sum_{j \in \mathcal{S}_i} w_j}{w_{app_i} + |\mathcal{S}_i| .w_{pre_i}} \\
T &= N_i \times \frac{w_{app_i} + |\mathcal{S}_i| .w_{pre_i}}{\sum_{j \in \mathcal{S}_i} w_j}
\end{aligned} \tag{2.5}$$

Hence the average computation time for one request of type i:

$$T^{server}_{comp_i} = \frac{w_{app_i} + |\mathcal{S}_i| . w_{pre_i}}{\sum_{j \in \mathcal{S}_i} w_j} \tag{2.6}$$

and the computation throughput for service i:

$$\rho^{comp}_{serv_i} = \frac{\sum_{j \in \mathcal{S}_i} w_j}{w_{app_i} + |\mathcal{S}_i| . w_{pre_i}} \tag{2.7}$$

Server communication model. A server of type i needs, for each request, to receive the request, and then to reply. Hence Equations (2.8) and (2.9) represent respectively the time to receive one request of type i, and the time to send the reply to its parent.

Communications time depends on the shape of the hierarchy, as the bandwidth is not the same over the platform. Thus, we denote by f^j_i the father of server $j \in \mathcal{S}_i$.

Let $T^{server_j}_{recv_i}$ be the receive time for one request of type i for server $j \in \mathcal{S}_i$:

$$T^{server_j}_{recv_i} = \frac{m_{req_i}}{B_{j,f^j_i}} \tag{2.8}$$

Let $T^{server_j}_{send_i}$ be the reply time for one request of type i for server $j \in \mathcal{S}_i$:

$$T^{server_j}_{send_i} = \frac{m_{resp_i}}{B_{j,f^j_i}} \tag{2.9}$$

Service throughput. Let ρ_{serv_i} be the service throughput. Concerning the machines model, and their ability to compute and communicate, we consider the following models:

– Send or receive or compute, single port: a node cannot do anything simultaneously.

$$\rho_{serv_i} = \frac{1}{\max_{j \in \mathcal{S}_i} \left\{ T^{server_j}_{recv_i} + T^{server_j}_{send_i} + T^{server}_{comp_i} \right\}} \tag{2.10}$$

– Send or receive, and compute, single port: a node can simultaneously send or receive a message, and compute.

$$\rho_{serv_i} = \min \left\{ \frac{1}{\max_{j \in \mathcal{S}_i} \left\{ T^{server_j}_{recv_i} + T^{server_j}_{send_i} \right\}}, \frac{1}{T^{server}_{comp_i}} \right\} \tag{2.11}$$

– Send, receive, and compute, single port: a node can simultaneously send and receive a message, and compute.

$$\rho_{serv_i} = \min \left\{ \frac{1}{\max_{j \in \mathcal{S}_i} \left\{ \max \left\{ T^{server_j}_{recv_i}, T^{server_j}_{send_i} \right\} \right\}}, \frac{1}{T^{server}_{comp_i}} \right\} \tag{2.12}$$

Agent model

We define the following variables for the agents. w_{req_i} is the amount of computation in $MFlops$ needed by an agent to process an incoming request of type i. Let $\mathcal{A}$ be the set of agents. For a given agent $A_j \in \mathcal{A}$, let $Chld^j_i$ be the set of children of A_j having service i in their underlying hierarchy. Also, let δ^j_i be a Boolean variable equal to 1 if and only if A_j has at least one child which knows service i in its underlying hierarchy. $w_{resp_i}\left(\left|Chld^j_i\right|\right)$ is the amount of computation in $MFlops$ needed to merge the replies of type i from its $\left|Chld^j_i\right|$ children. We suppose that this amount grows linearly with the number of children. This assumption is reasonable as we consider that only the best server is chosen, thus this work amounts to computing a maximum. Figure 2.3 presents the server model.

Our agent model relies on the underlying servers throughput. Hence, in order to compute the computation and communication times taken by an agent A_j, we need to know both the servers throughput ρ_{serv_i} for each $i \in \mathcal{R}$, and the children of A_j.

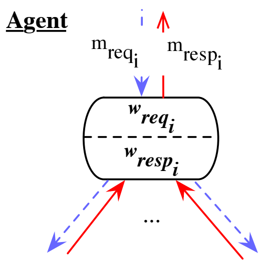


Figure 2.3: Agent model parameters.

Agent computation model. Let $T_{comp}^{agent_j}$ be the time for an agent A_j to schedule all requests it receives and forwards, when the servers provide a throughput ρ_{serv_i} for each $i \in \mathcal{R}$, it is given by Equation (2.13).

$$T_{comp}^{agent_j} = \frac{\sum_{i \in \mathcal{R}} \rho_{serv_i}.\delta_i^j.w_{req_i} + \sum_{i \in \mathcal{R}} \rho_{serv_i}.w_{resp_i} \left(\left| Chld_i^j \right| \right)}{w_j} \tag{2.13}$$

Agent communication model. Agent A_j needs, for each request of type i, to receive the request and forward it to the relevant children, then to receive the replies and forward the aggregated result back up to its parent. We also need to take into account the variation in the bandwidth between the agent and its children. Let f^j be the father of agent A_j in the hierarchy. Equations (2.14) and (2.15) present the time to receive and send all messages when the servers provide a throughput ρ_{serv_i} for each $i \in \mathcal{R}$.

Let $T_{recv}^{agent_j}$ be the receive time for agent $A_j \in \mathcal{A}$:

$$T_{recv}^{agent_j} = \sum_{i \in \mathcal{R}} \frac{\rho_{serv_i}.\delta_i^j.m_{req_i}}{B_{j,f^j}} + \sum_{i \in \mathcal{R}} \sum_{k \in Chld_i^j} \frac{\rho_{serv_i}.m_{resp_i}}{B_{j,k}} \tag{2.14}$$

Let $T_{send}^{agent_j}$ be the send time agent $A_j \in \mathcal{A}$:

$$T_{send}^{agent_j} = \sum_{i \in \mathcal{R}} \frac{\rho_{serv_i}.\delta_i^j.m_{resp_i}}{B_{j,f^j}} + \sum_{i \in \mathcal{R}} \sum_{k \in Chld_i^j} \frac{\rho_{serv_i}.m_{req_i}}{B_{j,k}} \tag{2.15}$$

We combine (2.13), (2.14), and (2.15) according to the chosen communication / computation model (similarly to Equations (2.10), (2.11), and (2.12)).

Lemma 2.3. *The highest throughput a hierarchy of agents is able to serve is limited by the throughput an agent having only one child of each kind of service can support.*

Proof. The bottleneck of such a hierarchy is clearly its root. Whatever the shape of the hierarchy, at its top, the root will have to support *at least* one child of each type of service (all messages have to go through the root). As the time required for an agent grows linearly with the number of children (see (2.13), (2.14) and (2.15)), having only one child of each type of service is the configuration that induces the lowest load on an agent. ■

2.3 Homogeneous Platform

We first consider a fully homogeneous, fully connected platform, *i.e.,* all nodes' processing power are the same w, and all links have the same bandwidth B, see Figure 2.4.

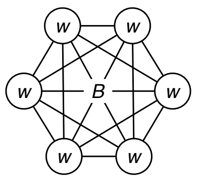


Figure 2.4: Homogeneous platform.

2.3.1 Model

As we consider a fully homogeneous platform, the equations for the model given in Section 2.2 can be simplified.

Server model. Let's consider that we have n_i servers of type i, and that n_i requests of type i are sent. On the whole, the n_i servers of type i require $\frac{n_i.w_{pre_i}+w_{app_i}}{w}$ time units to serve the n_i requests: each server has to compute the performance prediction n_i times, and serve one request. Hence, on average, the time to compute one request of type i is given by Equation (2.16).

$$T_{comp_i}^{server} = \frac{w_{pre_i} + \frac{w_{app_i}}{n_i}}{w} \tag{2.16}$$

Thus, the service throughput for requests of type i is given by the following formula, note that $\rho_{serv_i}^{comp}$ is the service throughput without taking into account communications:

$$\rho_{serv_i}^{comp} = \frac{w}{w_{pre_i} + \frac{w_{app_i}}{n_i}} \tag{2.17}$$

The communication model does not change. We only use B for the bandwidth.

Agent model. The agent computation model stays the same, we only use w for the computation power. However, the communication model can be slightly simplified. Equations (2.18) and (2.19) respectively present the receive and sent time for an agent.

$$T_{recv}^{agent_j} = \frac{\sum_{i \in \mathcal{R}} \rho_{serv_i}.\delta_i^j.m_{req_i} + \sum_{i \in \mathcal{R}} \rho_{serv_i}.\left|Chld_i^j\right|.m_{resp_i}}{B} \tag{2.18}$$

$$T_{send}^{agent_j} = \frac{\sum_{i \in \mathcal{R}} \rho_{serv_i}.\delta_i^j.m_{resp_i} + \sum_{i \in \mathcal{R}} \rho_{serv_i}.\left|Chld_i^j\right|.m_{req_i}}{B} \tag{2.19}$$

Remark. *Note that the above equations are consistent with the ones given in Section 2.2: if $\forall j, j' \in \mathcal{N}, B_{j,j'} = B$ and $\forall j \in \mathcal{N}, w_j = w$ in the general case formulae (Section 2.2), then we end up with the above formulae.*

2.3.2 Automatic Planning

Given the models presented in the previous section, we propose a heuristic for automatic deployment planning. The heuristic comprises two phases. The first step consists in dividing N nodes between the services, so as to support the servers. The second step consists in trying to build a hierarchy, with the $|V| - N$ remaining nodes, which is able to support the throughput generated by the servers. In this section, we present our automatic planning algorithm in three parts. We first present how the servers are allocated nodes, then a bottom-up approach to build a hierarchy of agents, and finally in Section 2.3.3 we present the whole algorithm.

Servers' repartition

The final goal of the deployment is to obtain for all services $i \in \mathcal{R}$ the same ratio $\frac{\rho_i}{\rho_i^*}$. However, as we try to build a hierarchy of agents being able to support the throughput offered by the servers, in the end we will have $\rho_i = \rho_{serv_i}$, ρ_{serv_i} being the service throughput, as defined in Section 2.2.1. Thus, we will try to have the same $\frac{\rho_{serv_i}}{\rho_i^*}$ ratio for all $i \in \mathcal{R}$. Algorithm 2.1 presents a simple way of dividing the available nodes to the different services. We iteratively increase the number of assigned nodes per services, starting by giving nodes to the service with the lowest $\frac{\rho_{serv_i}}{\rho_i^*}$ ratio.

Algorithm 2.1 Servers' repartition

```
Require: $N$: number of available nodes
Ensure: $n$: number of nodes allocated to the servers
1.1:  $S \leftarrow$ list of services in $\mathcal{R}$
1.2:  $n \leftarrow 0$
1.3:  repeat
1.4:      $i \leftarrow$ first service in $S$
1.5:      Assign one more node to $i$, and compute the new $\rho_{serv_i}$
1.6:      $n \leftarrow n + 1$
1.7:      if $\rho_{serv_i} \geq \rho_i^*$ then
1.8:          $\rho_{serv_i} \leftarrow \rho_i^*$
1.9:          $S \leftarrow S - \{i\}$
1.10:     end if
1.11:     $S \leftarrow$ Sort services by increasing $\frac{\rho_{serv_i}}{\rho_i^*}$
1.12: until $n = N$ or $S = \emptyset$
1.13: return $n$
```

Agents' hierarchy

Given the servers' repartition, and thus, the services' throughput ρ_{serv_i}, for all $i \in \mathcal{R}$, we need to build a hierarchy of agents that is able to support the throughput offered by the servers. Our approach is based on a bottom-up construction. We first distribute some nodes to the servers. Then, with the remaining nodes, we iteratively build levels of agents. Each level of agents has to be able to support the load incurred by the underlying level. The construction stops when only one agent is enough to support all the children of the previous level. In order to build each level, we make use of an integer linear program (ILP): ($\mathcal{LP}_1$).

We first need to define a few more variables. Let k be the current level: $k = 0$ corresponds to the server level. For $i \in \mathcal{R}$ let $n_i(k)$ be the number of elements (servers or agents) obtained at step k, which know service i. For $k \geq 1$, we recursively define new sets of agents. We define by M_k the number of available resources at step k: $M_k = M_1 - \sum_{l=1}^{k-1} \sum_{j=1}^{M_l} a_j(l)$. For $1 \leq j \leq M_k$ we define $a_j(k) \in \{0, 1\}$ to be a Boolean variable stating whether or not node j is an agent in step k: $a_j(k) = 1$ if and only if node j is an agent in step k. For $1 \leq j \leq M_k, \forall i \in \mathcal{R}, \delta_i^j(k) \in \{0, 1\}$ defines whether of not node j has service i in its underlying hierarchy in step k. At the server level, we have $k = 0$ (bottom of the hierarchy), M_0 is the number of nodes allocated to the servers. Then, $\delta_i^j(0)$ represents the fact that the node j is, or is not, a server of type i, thus we have for $1 \leq j \leq M_0, \forall i \in \mathcal{R}, \delta_i^j(0) = 1$ if and only if server j is of type i, otherwise $\delta_i^j(0) = 0$. Hence, we have the following relation: $\forall i \in \mathcal{R}, n_i(k) = \sum_{j=1}^{M_k} \delta_i^j(k)$. For $1 \leq j \leq M_k, \forall i \in \mathcal{R}, \left|Chld_i^j(k)\right| \in \mathbb{N}$ is as previously the number of children of node j which know service i. Finally, for $1 \leq j \leq M_k, 1 \leq l \leq M_{k-1}$ let $c_l^j(k) \in \{0, 1\}$ be a Boolean variable stating that node l in step $k - 1$ is a child of node j in step k: $c_l^j(k) = 1$ if and only if node l in step $k - 1$ is a child of node j in step k.

Using linear program ($\mathcal{LP}_1$), we can recursively define the hierarchy of agents, starting from the bottom of the hierarchy.

Let's have a closer look at ($\mathcal{LP}_1$). Lines (1), (2), and (3) only define the variables. Line (4) states that any element in level $k-1$ has to have exactly 1 parent in level k. Line (5) counts for each element at level k, its number of children that know service i. Line (6) states that the number of children of j of type i cannot be greater than the number of elements in level $k-1$ that know service i, and has to be 0 if $\delta_i^j(k) = 0$. The following two lines, (7) and (8), enforce the state of node j: if a node has at least a child, then it has to be an agent (line (7) enforces $a_j(k) = 1$ in this case), and conversely, if it has no children, then it has to be unused (line (8) enforces $a_j(k) = 0$ in this case). Line (9) states that at least one agent has to be present in the hierarchy. Line (10) is the transposition of the agent model in the *send or receive or compute, single port* model. Note that the other models can easily replace this model in ILP ($\mathcal{LP}_1$). This line states that the time required to deal with all requests going through an agent has to be lower than or equal to one second.

Finally, our objective function is the minimization of the number of agents: the equal share of obtained throughput to requested throughput ratio has already been cared of when allocating the nodes to the servers, hence our second objective that is the minimization of the number of agents in the hierarchy has to be taken into account.

$$
\begin{array}{l}
\textbf{Minimize } \displaystyle\sum_{j=1}^{M_k} a_j(k)\\
\textbf{Subject to}\\
\left\{
\begin{array}{lll}
(1) & 1 \le j \le M_k & a_j(k) \in \{0,1\}\\
(2) & 1 \le j \le M_k, \forall i \in \mathcal{R} & \delta_i^j(k) \in \{0,1\} \text{ and } \left|Chld_i^j(k)\right| \in \mathbb{N}\\
(3) & 1 \le j \le M_k, & \\
 & 1 \le l \le M_{k-1} & c_l^j(k) \in \{0,1\}\\
(4) & 1 \le l \le M_{k-1} & \displaystyle\sum_{j=1}^{M_k} c_l^j(k) = 1\\
(5) & 1 \le j \le M_k, \forall i \in \mathcal{R} & \left|Chld_i^j(k)\right| = \displaystyle\sum_{l=1}^{M_{k-1}} c_l^j(k).\delta_i^l(k-1)\\
(6) & 1 \le j \le M_k, \forall i \in \mathcal{R} & \left|Chld_i^j(k)\right| \le \delta_i^j(k).n_i(k-1)\\
(7) & 1 \le j \le M_k, i \in \mathcal{R} & \delta_i^j(k) \le a_j(k)\\
(8) & 1 \le j \le M_k & a_j(k) \le \displaystyle\sum_{i \in \mathcal{R}} \delta_i^j(k)\\
(9) & & \displaystyle\sum_{j=1}^{M_k} a_j(k) \ge 1\\
(10) & 1 \le j \le M_k & \sum_{i\in\mathcal{R}} \rho_{serv_i} \times\\
 & & \left(\dfrac{\delta_i^j(k).w_{req_i} + w_{resp_i}\left(\left|Chld_i^j(k)\right|\right)}{w} + \right.\\
 & & \dfrac{\delta_i^j(k).m_{req_i} + \left|Chld_i^j(k)\right|.m_{resp_i}}{B} +\\
 & & \left. \dfrac{\delta_i^j(k).m_{resp_i} + \left|Chld_i^j(k)\right|.m_{req_i}}{B} \right) \le 1
\end{array}
\right.
\end{array}
\qquad (\mathcal{LP}_1)
$$

Note that ($\mathcal{LP}_1$) is presented for a sequential model. Constraint (10) in ($\mathcal{LP}_1$) can be adapted to fit to computation/communication parallel model, or totally parallel model.

Remark. *In order to improve the converge time to an optimal solution for linear program ($\mathcal{LP}_1$), we can add the following constraint:*

$$a_1(k) \ge a_2(k) \cdots \ge a_{M_k}(k) \tag{2.20}$$

This states that only the first nodes can be agents. This prevents the solver from trying all swapping possibilities when searching for a solution. We can safely add this constraint, as we suppose that we have a homogeneous platform.

2.3.3 Building the Whole Hierarchy

So far, we did not talk about the repartition of the available nodes between agents and servers. We will now present the whole algorithm for building the hierarchy.

Maximum attainable throughput per service. Whatever the expected throughput for each service is, there is a limit on the maximum attainable throughput. Given Equations (2.13), (2.18), and (2.19), and the fact that a hierarchy must end at the very top by only one agent, the maximum throughput attainable by an agent serving all kinds of services (which is the case of the root of the hierarchy), is attained when the agent has only one child of each service (see Lemma 2.3). Hence, the maximum attainable throughput for each service, when all services receive the same served to required throughput ratio $\frac{\rho_i}{\rho_i^*}$, from the agents' point of view is given by linear program ($\mathcal{LP}_2$) which computes $\rho_{serv_i}^{max}$ for $i \in \mathcal{R}$, the maximum attainable throughput for each service i that an agent can offer under the assumption that all services receive an equal share.

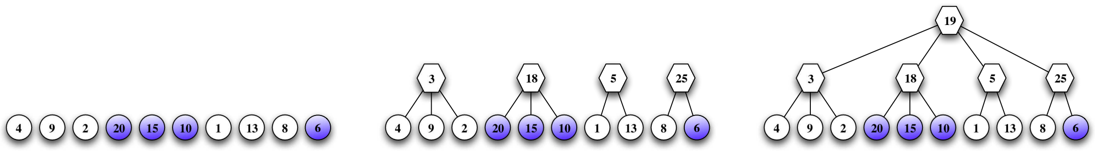
$$\begin{array}{l}
\textbf{Maximize } \mu \\
\textbf{Subject to} \\
\left\{\begin{array}{lll}
(1) & \forall i \in \mathcal{R} & \mu \leq \dfrac{\rho_{serv_i}^{max}}{\rho_i^*} \text{ and } \mu \in [0,1], \rho_{serv_i}^{max} \in [0, \rho_i^*] \\
(2) & \forall i, i' \in \mathcal{R} & \dfrac{\rho_{serv_i}^{max}}{\rho_i^*} = \dfrac{\rho_{serv_{i'}}^{max}}{\rho_{i'}^*} \\
(3) & 1 \leq j \leq M_k & \sum_{i \in \mathcal{R}} \rho_{serv_i}^{max} \times \\
& & \left(\dfrac{w_{req_i} + w_{resp_i}}{w} + \dfrac{2.m_{req_i} + 2.m_{resp_i}}{B} \right) \leq 1
\end{array}\right.
\end{array} \qquad (\mathcal{LP}_2)$$

When building the hierarchy, there is no point in allocating nodes to a service i if ρ_{serv_i} gets higher than $\rho_{serv_i}^{max}$. Hence, whenever a service has a throughput higher than $\rho_{serv_i}^{max}$, then we consider that its value is $\rho_{serv_i}^{max}$ when building the hierarchy. Thus, lines 1.7 and 1.8 in Algorithm 2.1 become:

```
1.7: if $\rho_{serv_i}^{comp} \geq \min\left\{\rho_i^*, \rho_{serv_i}^{max}\right\}$ then
1.8:     $\rho_{serv_i} \leftarrow \min\left\{\rho_i^*, \rho_{serv_i}^{max}\right\}$
1.9: end if
```

Building the hierarchy. Algorithm 2.2 presents how to build a hierarchy, it proceeds as follows. We first try to give as many nodes as possible to the servers (line 2.4 to 2.7), and we try to build a hierarchy on top of those servers with the remaining nodes (line 2.8 to 2.29). Whenever building a hierarchy fails, we reduce the number of nodes available for the servers (line 2.28, note that we can use a binary search to reduce the complexity, instead of decreasing one by one the number of available nodes). Hierarchy construction may fail for several reasons: no more nodes are available for the agents (line 2.10), ($\mathcal{LP}_1$) has no solution (line 2.12), or only *chains* of agents have been built, *i.e.,* each new agent has only one child (line 2.22). If a level contains agents with only one child, those nodes are set as available for the next level, as having chains of agents in a hierarchy is useless (line 2.26). Finally, either we return a hierarchy if we found one, or we return a hierarchy with only one child of each type $i \in \mathcal{R}$, as this means that the limiting factor is the hierarchy of agents. Thus, only one server of each type of service is enough, and we cannot do better than having only one agent. Figure 2.5 presents an example of our bottom up approach.

Correcting the throughput. Once the hierarchy has been computed, we need to correct the throughput for services that were limited by the agents. Indeed, the throughput computed using ($\mathcal{LP}_2$) may be too restrictive for some services. The values obtained implied that we had effectively an equal ratio between obtained throughput over requested throughput for all services, which may not be the case if a service requiring lots of computation is deployed alongside a service requiring very few computation. Hence, once the hierarchy is created, we need to compute what is really the throughput that can be obtained for each service on the hierarchy. To do so, we simply use our agent model, with fixed values for ρ_{serv_i} for all $i \in \mathcal{R}$ such that the throughput of i is not limited by the agents, and we try to maximize



(a) Nodes repartition for servers (b) First level of agents (c) Final level of agents

Figure 2.5: Bottom-up approach.

Algorithm 2.2 Build hierarchy

```
2.1:  $N \leftarrow |V| - 1$ {One node for an agent, $|V| - 1$ for the servers}
2.2:  $Done \leftarrow$ false
2.3:  while $N \geq |\mathcal{R}|$ and not $Done$ do
2.4:    Use Algorithm 2.1 to find the server repartition with $N$ nodes
2.5:    $nbUsed \leftarrow$ number of nodes used by Algorithm 2.1
2.6:    $M_0 \leftarrow nbUsed$
2.7:    Set all variables: $n_i(0)$, $a_j(0)$, $\delta_i^j(0)$, $Chld_i^j(0)$ and $c_l^j(0)$
2.8:    $k \leftarrow 1$
2.9:    $M_k \leftarrow |V| - nbUsed$
2.10:   while $M_k > 0$ and not $Done$ do
2.11:     Compute level $k$ using linear program ($\mathcal{LP}_1$)
2.12:     if level $k$ could not be built (i.e., ($\mathcal{LP}_1$) failed) then
2.13:       break
2.14:     end if
2.15:     $nbChains \leftarrow$ count the number of agents having only 1 child
2.16:     $availNodes \leftarrow M_k$
2.17:     $M_k \leftarrow \sum_{j=1}^{M_k} a_j(k)$ {Get the real number of agents}
2.18:     if $M_k == 1$ then
2.19:       $Done \leftarrow$ true {We attained the root of the hierarchy}
2.20:       break
2.21:     end if
2.22:     if $nbChains == M_{k-1}$ then
2.23:       break{This means we added 1 agent over each element at level $k - 1$}
2.24:     end if
2.25:     $k \leftarrow k + 1$
2.26:     $M_k \leftarrow availNodes - M_{k-1} + nbChains$
2.27:   end while
2.28:   $N \leftarrow nbUsed - 1$
2.29: end while
2.30: if $Done$ then
2.31:   return the hierarchy built with ($\mathcal{LP}_1$) without chains of agents
2.32: else
2.33:   return a star graph with one agent and one server of each type $i \in \mathcal{R}$
2.34: end if
```

the values of ρ_{serv_i} for all services that are limited by the agents. We use linear program ($\mathcal{LP}_3$) and Algorithm 2.3 for this purpose.

Equation (1) states that μ is the minimum of all ratios, and that the value of ρ_i^{max} cannot be greater than the throughput that is offered at the server level ρ_{serv_i}. The following equation, equation (2), ensures that available bandwidth and computing power are not violated.

Algorithm 2.3 Correct Throughput

```
3.1: $\mathcal{R}_{agLim} \leftarrow i \in \mathcal{R}$ such that service $i$ is "agent limited"
3.2: $Ag \leftarrow$ set of agents per level
3.3: while $\mathcal{R}_{agLim} \neq \emptyset$ do
3.4:     $\mu, \{\rho_i^{max}\} \leftarrow$ Solve linear program ($\mathcal{LP}_3$)
3.5:     Find $i \in \mathcal{R}_{agLim}$ such that $\mu = \frac{\rho_i^{max}}{\rho_i^*}$
3.6:     $\rho_{serv_i} \leftarrow \rho_i^{max}$
3.7:     $\mathcal{R}_{agLim} \leftarrow \mathcal{R}_{agLim} - \{i\}$
3.8: end while
```

Maximize μ
Subject to

$$\left\{\begin{array}{lll}
(1) & \forall i \in \mathcal{R}_{agLim} & \mu \in [0,1], \mu \leq \dfrac{\rho_i^{max}}{\rho_i^*} \text{ and } 0 \leq \rho_i^{max} \leq \rho_{serv_i} \\
(2) & \forall k, \forall j \in Ag[k] & \displaystyle\sum_{i \in \mathcal{R}-\mathcal{R}agLim} \rho_{serv_i} . \frac{\delta_i^j(k).w_{req_i} + \left|Chld_i^j(k)\right|.w_{resp_i}}{w} + \\
 & & \displaystyle\sum_{i \in \mathcal{R}agLim} \rho_i^{max} . \frac{\delta_i^j(k).w_{req_i} + \left|Chld_i^j(k)\right|.w_{resp_i}}{w} + \\
 & & \displaystyle\sum_{i \in \mathcal{R}-\mathcal{R}agLim} \rho_{serv_i} . \frac{\delta_i^j(k).m_{req_i} + \left|Chld_i^j(k)\right|.m_{resp_i}}{B} + \\
 & & \displaystyle\sum_{i \in \mathcal{R}agLim} \rho_i^{max} . \frac{\delta_i^j(k).m_{req_i} + \left|Chld_i^j(k)\right|.m_{resp_i}}{B} + \\
 & & \displaystyle\sum_{i \in \mathcal{R}-\mathcal{R}agLim} \rho_{serv_i} . \frac{\delta_i^j(k).m_{resp_i} + \left|Chld_i^j(k)\right|.m_{req_i}}{B} + \\
 & & \displaystyle\sum_{i \in \mathcal{R}agLim} \rho_i^{max} . \frac{\delta_i^j(k).m_{resp_i} + \left|Chld_i^j(k)\right|.m_{req_i}}{B} \leq 1
\end{array}\right. \qquad (\mathcal{LP}_3)$$

2.3.4 Comparing DIET and the Model

In this section, we confront our model with our target middleware, DIET, over the Grid'5000 platform.

We will denote by *dgemm* x the call to a dgemm service on matrices of size $x \times x$, and *Fibonacci* x the call to a Fibonacci service to compute the Fibonacci number for $n = x$.

Request forwarding

The model supposes that an agent only receives a request if it knows in its underlying hierarchy this service. This assumption is partly verified, as in fact, the root of the hierarchy (the Master Agent, MA) receives all the clients requests, even if the requested service is not present in its underlying hierarchy. However, this is true for all internal agents (the Local Agents, LA), as a request is forwarded to a child if and only if this child knows the service.

Note however, that in our study, as we require that all requested services are present in the hierarchy (*i.e.,* at least one SED of each service $i \in \mathcal{R}$ exists in the hierarchy), all services are present in the MA underlying hierarchy.

Messages

Our model assumes that the message sizes do not increase as they travel down and up the hierarchy. This is the case when an agent selects only one server whenever a response is sent back. By default, DIET returns the 10 best servers. However, an option allows the client to specify the maximum server list size it wishes to receive. Hence, for the experiments presented in this paper we set the size of the list to one.

Our model can easily be extended to the case where an agent returns the complete list of servers found in its underlying hierarchy. If we suppose that reply messages in the hierarchy grow linearly with the number of servers found so far. Up to now, we considered that each agent replied only with the choice of one server, whatever the actual number of servers in the underlying hierarchy. The middleware could also return to the client the complete list of servers. Hence, at each level of the hierarchy the reply message would grow. This can easily be taken into account in our model. Let $Cpt_i^j(k)$ be the number of servers of type i in the hierarchy under element j at level k. For level $k = 0$ we set $Cpt_i^j(0) = 1$ if node j is a server of type i, otherwise we set $Cpt_i^j(0) = 0$. For $k > 0, i \in \mathcal{R}, 1 \le j \le M_k$ we have $Cpt_i^j(k) = \sum_{l=1}^{M_{k-1}} Cpt_i^l(k-1) \times c_l^j(k)$.

Then, we would need to change the equations for the sending and receiving time at the agent level with the following ones:

$$\frac{\sum_{i\in\mathcal{R}} \rho_{serv_i}.\delta_i^j(k).m_{req_i} + \sum_{i\in\mathcal{R}} \rho_{serv_i}.m_{resp_i}.\sum_{l=1}^{M_{k-1}} c_l^j(k).Cpt_i^l(k-1)}{B} \tag{2.21}$$

$$\frac{\sum_{i\in\mathcal{R}} \rho_{serv_i}.m_{resp_i}.\sum_{l=1}^{M_{k-1}} c_l^j(k).Cpt_i^l(k-1) + \sum_{i\in\mathcal{R}} \rho_{serv_i}.\left|Chld_i^j(k)\right|.m_{req_i}}{B} \tag{2.22}$$

Agent model

Our model requires that the scheduling cost for a request i at an agent level be linear in the number of children of type i. Figure 2.6 presents the scheduling time for an agent versus the number of children of type i obtained with DIET. As can be seen the linear cost assumption is verified.

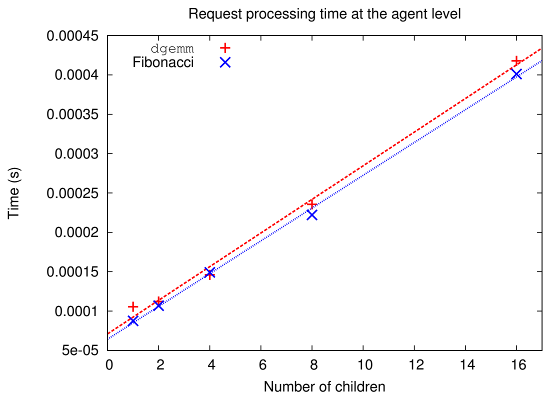
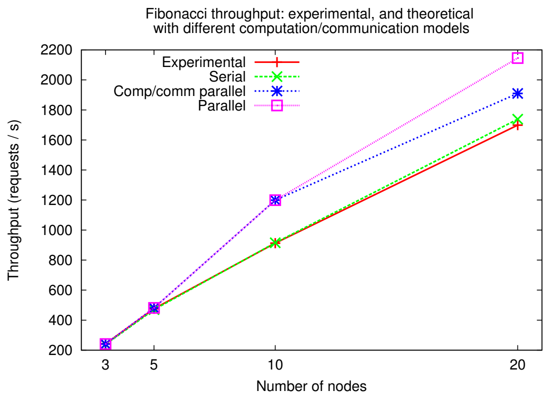


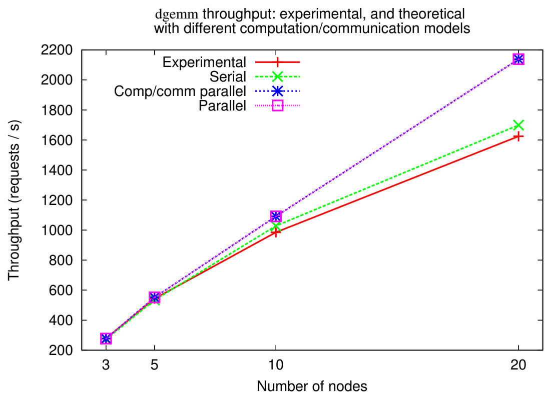
Figure 2.6: Mean time required per request versus the number of children.

Computation/communication models versus experiments

In Figures 2.7a and 2.7b, we present a comparison between the theoretical computation/communication models presented in Section 2.2 and experimental throughput for Fibonacci and `dgemm` services. As can be seen the *serial* model is the one that best fits the experimental results.



(a) Fibonacci



(b) `dgemm`

Figure 2.7: Services experimental and theoretical throughput with the different communication/computation models.

2.3.5 Benchmarking

We need to benchmark the communication and computation performance of the platform, and the communication and computation requirements of the DIET middleware.

We realized our experiments on a cluster of the French experimental platform: Grid'5000. We used a 79-nodes cluster, the Sagittaire cluster from the Lyon site. Each node has an AMD Opteron 250 CPU at 2.4GHz, with 1MB of cache and 2GB of memory. All these nodes are connected on a Gigabit Ethernet network supported by an Extreme Networks Blackdiamond 8810. The operating system deployed on the Sagittaire nodes was a Debian 2.6.24.3 x86_64. Finally, the chosen CORBA implementation was omniORB 4.1.2 [103], as DIET relies on CORBA for communications; and we used gcc 4.3.2. We used the Atlas library [165] version of `dgemm`.

Communications

In order to analyze the communications within a DIET hierarchy, we have a simple hierarchy composed of one MA, one LA and one SED for a given service. Then, a client requests the service, and using `tcpdump` [124] we retrieve the communication traces. Finally, using `tcptrace` [135] to analyze those traces, we obtain the communication requirements for our model: m_{req_i} and m_{resp_i}. This approach provides a measurement of the entire message size, including headers. Results are given in Table 2.1.

Service	m_{req_i}	m_{resp_i}
`dgemm`	5.016	5.328
Fibonacci	4.078	5.328

Table 2.1: Communication requirements. Message size is given in kilobit.

Given the communication requirements, and the size of the messages, we then determine the bandwidth the platform can offer. Using NWS [166], we obtain a bandwidth of $B = 186.7 Mbit/s$.

Computation

In order to determine the SED computation parameters, we deploy a simple platform composed of one MA, one LA and one SED. Then, we run as many clients as necessary to completely load the platform during 60 seconds. During this 60 seconds run, we measure the time required by the SED to predict its performance, and to compute the service. This method allows us to obtain a mean value on the prediction and service times. We benchmark two services: a matrix multiplication using `dgemm` [83] (for various matrix sizes), and the computation of the Fibonacci number using a naive algorithm.

For the agent parameters, as w_{resp_i} depends on the number of children, we deploy a variety of star deployments (one MA and several SEDs), then we load the platform with enough clients for 60 seconds, and measure the time taken for processing an incoming request, and the time taken for scheduling the replies. The scheduling applied at the agent level is a simple Round-Robin.

The number of clients used for benchmarking in order to correctly load the platform is determined as follows. For each platform to benchmark, we launch a new client every second, during 400 seconds. A client can send to DIET only one request at a given time, but does so in an infinite loop. Then, we analyze the obtained throughput versus the number of clients, and determine what is the optimal number of clients, *i.e.,* the number of clients that allows to obtain the highest throughput. Figure 2.8 presents such an execution. On this figure, the maximum is obtained at around 100s, thus 100 clients. With more clients, we can see that the throughput is less stable.

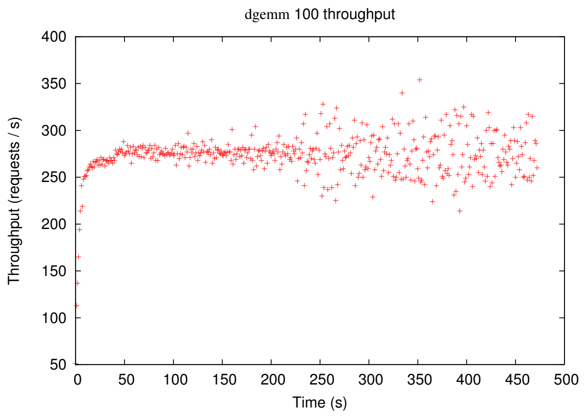


Figure 2.8: Throughput analysis to find the suitable number of clients to load the platform.

Finally, we measure the computing power of our machines using benchmarks extracted from High Performance Linpack [141] (HPL), compiled with the Atlas [164] version of BLAS. We found a value of $w = 3.249$ GFlops. We use this value to convert the previously obtained times into required MFlops. Table 2.2 gives the obtained SED values for all the services used in the experimental section, and Table 2.3 gives the obtained agent values.

Service	w_{pre_i}	w_{app_i}
`dgemm` 10	0.078	0.603
`dgemm` 100	0.063	11.657
`dgemm` 500	0.165	363.505
Fibonacci 20	0.047	0.591
Fibonacci 30	0.021	13.444
Fibonacci 40	0.008	1695.010

Table 2.2: Mean SED w_{pre_i} and w_{app_i} values for `dgemm` and Fibonacci services (in MFlops).

Service	w_{req_i}	w_{resp_i}
`dgemm`	0.230	0.069
Fibonacci	0.235	0.068

Table 2.3: Mean agent w_{req_i} and w_{resp_i} parameters for `dgemm` and Fibonacci services (in MFlops).

Once all the above parameters were determined, we then compared the throughput given by our model with experimental results.

2.3.6 Experimental Results

In this section, we present comparisons between the theoretical predictions and real experiments with the DIET middleware. We also show that our bottom-up approach based on an ILP to build the hierarchy gives really good results compared to classical deployments.

Theoretical model / experimental results comparison

In order to validate our model, we conducted experiments with DIET on the French experimental testbed Grid'5000. Using the benchmark values obtained in Section 2.3.5, and our bottom-up algorithm, we generated for various combinations of `dgemm` and Fibonacci services, for platforms with a number of nodes ranging from 3 to 50. Even though the algorithm is based on an ILP, it took only a few seconds to generate the 35 hierarchies for our experiments. To solve the ILP, we relied on ILOG CPLEX. Deployments of the DIET middleware have been done using the "historical DIET deployment software": GoDIET [58]. However, we also added the possibility to deploy DIET with two other deployment software: ADAGE [119] and TUNe [54]. ADAGE provides faster deployments of DIET than with GoDIET; and TUNe also provides basic autonomic management.

Our goal here is to stress DIET, so we use relatively "small" services, *i.e.,* calls to services with relatively few computations. We compared the throughput measured and predicted for various services combinations: `dgemm` 100×100 and Fibonacci 30 (medium size services), `dgemm` 10×10 and Fibonacci 20 (small size services), `dgemm` 10×10 and Fibonacci 40 (small size `dgemm`, large size Fibonacci), `dgemm` 500×500 and Fibonacci 20 (large size `dgemm`, small size Fibonacci), `dgemm` 500×500 and Fibonacci 40 (large size `dgemm`, large size Fibonacci).

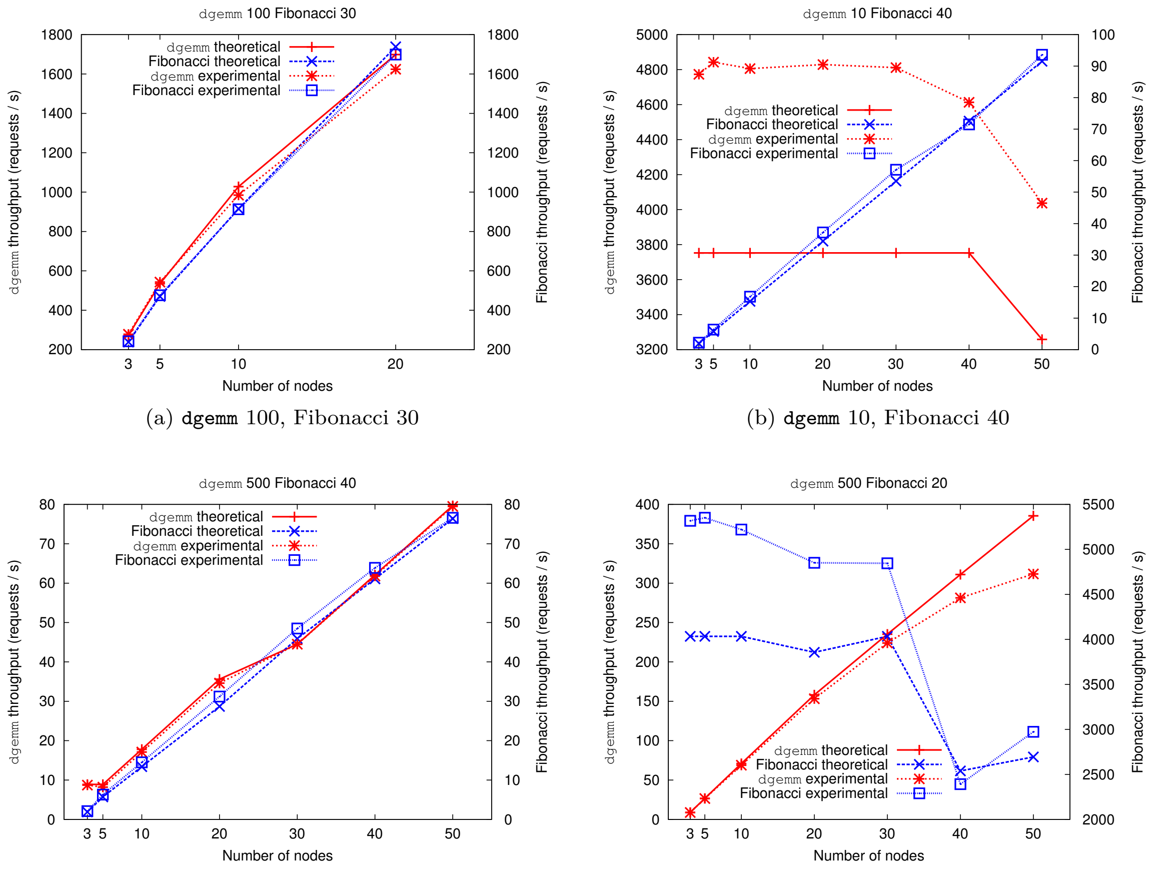
We need to set a target throughput ρ_i^* for each service. We asked for the same throughput for each service: $\forall i \in \mathcal{R}$, $\rho_i^* = 10000$. Hence, whenever the algorithm has enough nodes, we should end up with the same throughput for each service, and thus the same $\frac{\rho_i}{\rho_i^*}$ ratio. We intentionally selected a high value for ρ_i^* in order to use as many nodes as possible.

The experimental protocol is the following. We first deploy the hierarchy with GoDIET, then we run clients for both services and wait 10 seconds so that the load stabilizes. Finally, we measure the throughput during 60 seconds before killing all clients and the DIET hierarchy. Each client runs on a separate node, and sends one request at a time, once a request is completed, a new one is sent, this is repeated indefinitely.

Table 2.4 presents the relative error between the theoretical and experimental throughput: a dash in the table means that the algorithm returned the same hierarchy as with fewer nodes, and hence the results are the same. Figures 2.9a, 2.9b, 2.9c and 2.9d compare the throughput obtained with our model and during our experiments.

As can be seen, the experimental results closely follow what the model has predicted. Higher errors can be observed for really small services (`dgemm` 10 and Fibonacci 20). This is due to the fact that really small services are harder to benchmark: the obtained w_{pre_i} and w_{app_i} values are higher than what they should be. Note however that even if theoretical throughputs for small services are lower than experimental ones, the model correctly detects when the throughput drops due to the load at the agent level (see Figures 2.9b and 2.9d). Figure 2.9a does not extend to 50 nodes, this is due to the fact that our algorithm returned exactly the same hierarchy for more than 20 nodes. Adding new servers to this hierarchy did not improve the performance.

In Figure 2.9d, we can observe that `dgemm` performance degrades as the number of nodes increases. This seems to come from CORBA communications. As we explain in Section 2.6, modifying CORBA options increased greatly the obtained throughput. However, we did not manage to make the experiments exactly match the model with `dgemm` 500, Fibonacci 20 experiments.



(a) `dgemm` 100, Fibonacci 30

(b) `dgemm` 10, Fibonacci 40

(c) `dgemm` 500, Fibonacci 40

(d) `dgemm` 500, Fibonacci 20

Figure 2.9: Comparison theoretical/experimental throughput.

Experiment		**Number of nodes**						
		3	**5**	**10**	**20**	**30**	**40**	**50**
`dgemm` 100, Fibonacci 30	`dgemm`	1.87%	1.58%	4.17%	4.39%	-	-	-
	Fibonacci	1.76%	1.26%	0.26%	2.28%	-	-	-
`dgemm` 10, Fibonacci 20	`dgemm`	27.4%	-	-	-	-	-	-
	Fibonacci	27.3%	-	-	-	-	-	-
`dgemm` 10, Fibonacci 40	`dgemm`	27.2%	29.1%	28.1%	28.7%	28.2%	22.9%	23.9%
	Fibonacci	14.6%	10.4%	9.7%	8.0%	6.7%	1.4%	2.3%
`dgemm` 500, Fibonacci 20	`dgemm`	1.1%	0.7%	3.0%	3.0%	4.7%	9.5%	19.1%
	Fibonacci	31.8%	32.6%	26.4%	25.8%	20.1%	5.8%	10.4%
`dgemm` 500, Fibonacci 40	`dgemm`	2.2%	7.9%	3.9%	2.8%	0.0%	0.5%	0.3%
	Fibonacci	10.5%	10.5%	8.2%	8.7%	5.7%	4.6%	0.3%

Table 2.4: Experimental throughput: relative error.

Relevancy of creating new agent levels

In order to validate the relevancy of our algorithm to create hierarchies, we compared the throughputs obtained with our hierarchies, and the ones obtained with a star graph having exactly the same allocation of servers obtained with our algorithm. Thus, we aim at validating the fact that our algorithm creates several levels of agents whenever required. In fact in the previous experiments, this happens only for the following deployments:

– `dgemm` 100, Fibonacci 30 on 20 nodes: 1 MA and 3 LA,
– `dgemm` 500, Fibonacci 20 on 30, 40 and 50 nodes: 1 MA and 2 LA.

It is clear that with more nodes, the number of levels would increase. Moreover, in our examples we only rely on basic scheduling done by DIET (a sort of Round-Robin) which does not require much computation at the agent level. With a more complex scheduling algorithm, the load at the agent level would be increased, and hence more agent levels would be created.

Here are the results we obtained with such star graphs. We present the gains/losses obtained by the star graph deployments compared to the results obtained with the hierarchies our algorithm computed:

– **`dgemm` 100, Fibonacci 30 on 20 nodes:** we obtained a throughput of 912 requests per seconds for `dgemm` and 779 requests per seconds for Fibonacci. Hence a loss of 43.8% and 54.1% respectively.
– **`dgemm` 500, Fibonacci 20 on 30 nodes:** we obtained a throughput of 212 requests per seconds for `dgemm` and 3491 requests per seconds for Fibonacci. Hence a loss of 5.5%, and 28% respectively.
– **`dgemm` 500, Fibonacci 20 on 40 nodes:** we obtained a throughput of 238 requests per seconds for `dgemm` and 2477 requests per seconds for Fibonacci. Hence a loss of 15.4% for `dgemm`, and a gain of 3.6% for Fibonacci.
– **`dgemm` 500, Fibonacci 20 on 50 nodes:** we obtained a throughput of 185 requests per seconds for `dgemm` and 2494 requests per seconds for Fibonacci. Hence a loss of 40.7% and 16.1% respectively.

We can see from the above results that our algorithm creates new levels of agents whenever required. Without the new agent levels, the obtained throughput can be much lower. This is due to the fact that the MA becoming overloaded, and thus does not have enough time to schedule all requests.

Relevancy of the servers allocation

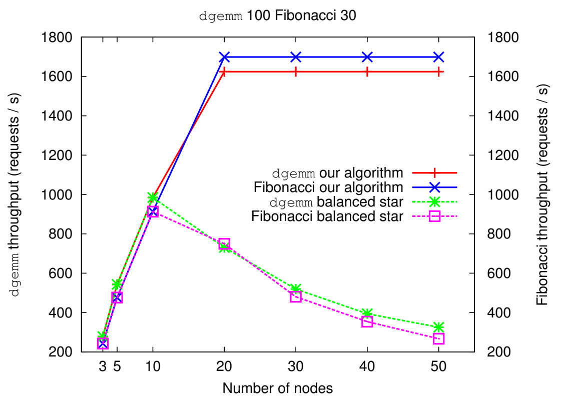
We also compared the throughput obtained by our algorithm, and the one by a balanced star graph (*i.e.,* a star graph where all services received the same number of servers). A balanced star graph is the naive approach that is generally used when the same throughput is requested for all services, which is what we aimed at in our experiments. Figures 2.10a to 2.10e present the comparison between the throughput obtained with both methods. Our algorithm gives better results. The throughput is better on all but one experiment, and no more resources than necessary are used (in Figure 2.10a, no more than 20 nodes are required to obtain the best throughput, and in Figure 2.10b only 3 nodes are used).

The only experiment where the balanced star graph produced a higher throughput is for `dgemm` 500 in the `dgemm` 500 Fibonacci 40 experiment (see Figure 2.10e). However, this is at the expense of the second service which has a lower throughput.

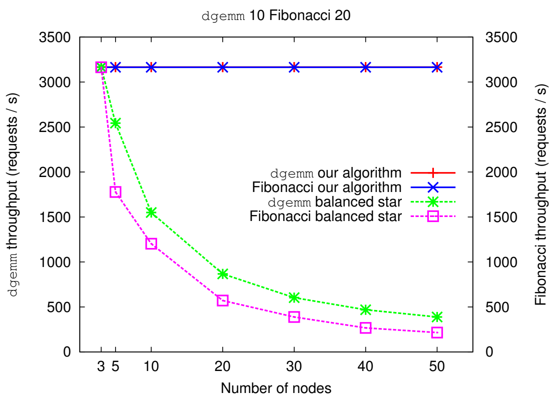
The `dgemm` throughput loss observable in Figure 2.10d is due to the load at the agent level (the same phenomena is observable in Figure 2.9d and is well predicted by our model).

Using our algorithm, we are able to increase the throughput to up to 700% (see Figure 2.10b for instance). It also tries to balance the $\frac{\rho_i}{\rho_i^*}$ ratio without degrading the performance, whereas with the balanced star graph adding more nodes can degrade the performance of both services.

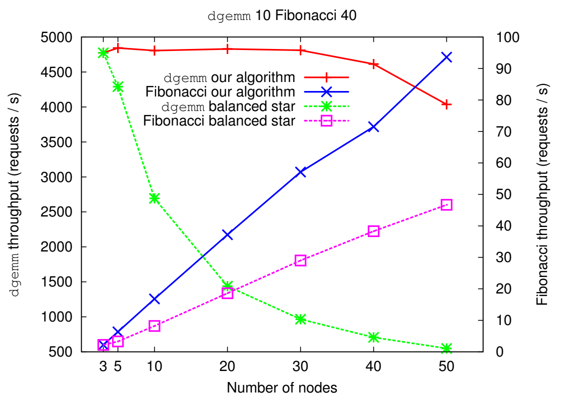
In this section, we saw that our model can accurately predict the behavior a hierarchical middleware, and that our bottom-up approach to build a hierarchy gives much better results than classical deployment approaches. However, fully homogeneous platforms tend to disappear to become heterogeneous platforms. Thus, we need to extend our algorithms to take into account heterogeneity.



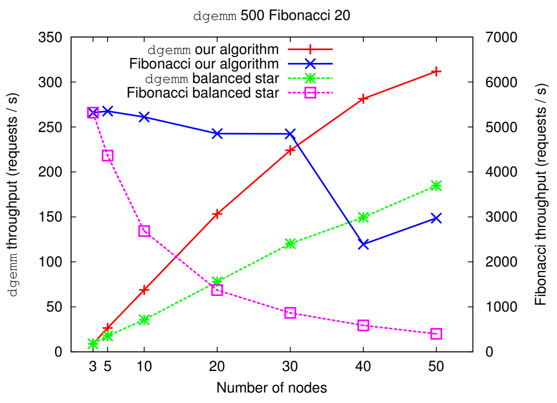
(a) **dgemm** 100, Fibonacci 30



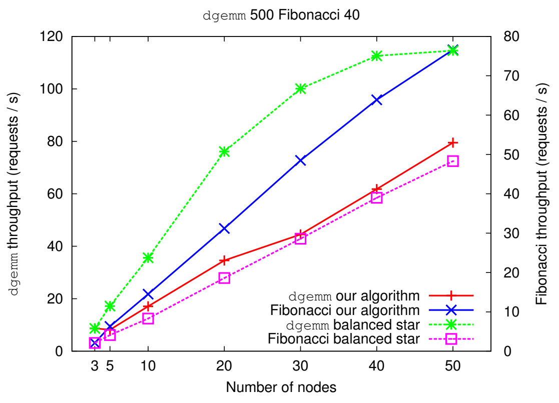
(b) **dgemm** 10, Fibonacci 20



(c) **dgemm** 10, Fibonacci 40



(d) **dgemm** 500, Fibonacci 20



(e) **dgemm** 500, Fibonacci 40

Figure 2.10: Comparison: our algorithm and balanced star.

2.4 Heterogeneous Computation, Homogeneous Communication

Before moving to a fully heterogeneous platform, we first deal with a simpler case. We assume that nodes' computing power w_j can be heterogeneous, but that communication links are homogeneous with a bandwidth B. This type of platform is quite close to a fully homogeneous one, and thus does not discard our bottom-up approach presented in Section 2.3. In fact, we only need to adapt the Linear Program ($\mathcal{LP}_1$), and Algorithm 2.1 to take into account the computing power heterogeneity.

2.4.1 Algorithm

We do not present the whole linear program, as it is exactly the same as ($\mathcal{LP}_1$), the only difference being that in constraint (10) w becomes w_j. We will refer to this modified Linear Program by ($\mathcal{LP}_4$).

We also use the same approach to distribute the nodes for the servers. Thus, Algorithm 2.1 can be reused. The only modification being that we need to take into account the nodes' heterogeneity. Hence, we propose two heuristics: *min-first* which first give the less powerful nodes to the servers, and *max-first* which first give the more powerful nodes to the servers.

Finally, Algorithm 2.2 can also be reused, we only need to keep track of which nodes have been used as they do not all have the same characteristics anymore.

All in all, we use the same schema as in Section 2.3: first compute the maximum supported throughput by an agent (we use for this the most powerful node), then distribute some nodes to the servers using either *min-first* or *max-first* heuristics, then try to build a hierarchy of agents using ($\mathcal{LP}_4$). Once a hierarchy is found, we correct the obtained throughput.

2.4.2 Experiments

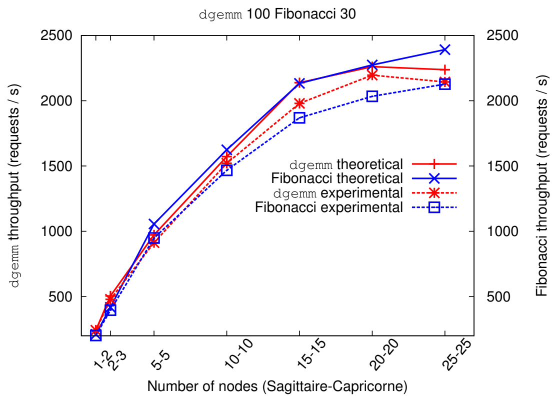
We ran experiments on DIET hierarchies containing two services: `dgemm` 100 and Fibonacci 30. We used two clusters on the Lyon site of Grid'5000: Sagittaire and Capricorne. Their respective computing power is $w_{sagittaire} = 3249 MFlops$ and $w_{capricorne} = 2922 MFlops$. We used two strategies to sort the nodes that are divided between the services: either we sorted them by increasing w_j (heuristic *min-first*) or by decreasing w_j (heuristic *max-first*).

Theoretical model / experimental results comparison

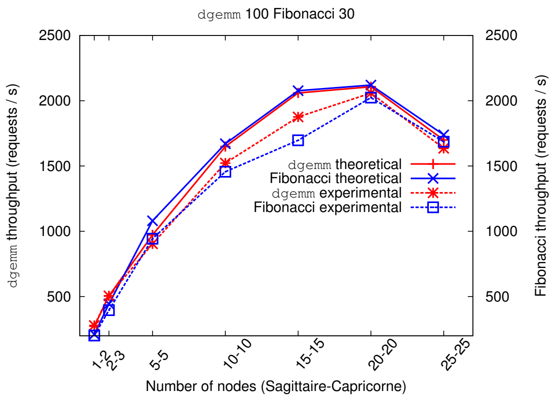
As has been done for the homogeneous platform case, we validate our model against real executions with the DIET middleware. Figures 2.11a and 2.11b present the comparison between theoretical and experimental throughput for respectively experiments with *min-first* and *max-first* heuristics. Table 2.5 presents the relative error for those experiments. As can be seen, the *min-first* heuristic is the most interesting one when dealing with large platforms. This can easily be explained. Using less powerful nodes for servers leaves more powerful nodes for the agents, hence the maximum attainable throughput due to agents limitation is higher. Whatever the heuristic, we remain within 18.3% of the theoretical model.

Experiment		**Number of nodes**						
		1-2	**2-3**	**5-5**	**10-10**	**15-15**	**20-20**	**25-25**
min-first	`dgemm`	0.3%	5.2%	6.1%	3.2%	3.9%	2.9%	4.20%
	Fibonacci	4.1%	5.1%	10.2%	9.7%	10.0%	10.6%	11.0%
max-first	`dgemm`	2.1%	0.1%	7.0%	7.7%	8.9%	2.3%	3.8%
	Fibonacci	4.6%	10.8%	12.7%	12.8%	18.3%	4.5%	3.0%

Table 2.5: Relative error, using *min-first* and *max-first* heuristics.



(a) *min-first*



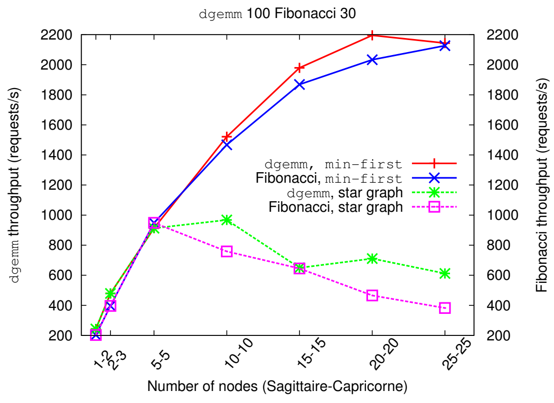
(b) *max-first*

Figure 2.11: **dgemm** 100, Fibonacci 30 theoretical and experimental throughput, with *min-first* and *max-first* heuristics.

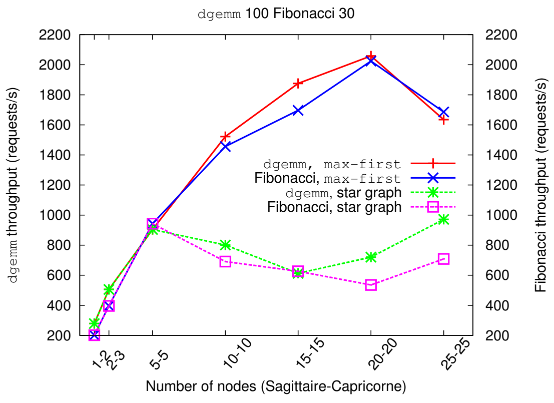
Comparison with star graphs

On the tested platforms, our heuristics created hierarchies having up to three levels of agents. Is this really necessary, or would have a star graph, having the same SeD mapping, given the same or better results? Figure 2.12a presents the comparison between *min-first* and a star graphs having the same SeD distribution, and Figure 2.12b presents the comparison between *max-first* and star graphs having the same SeD distribution. As can be seen, our approach clearly surpasses the simple star graph approach.

The fact that on Figure 2.12b the star graph throughput increases with 25-25 nodes is easily explained by the number of SeDs this star graph has. As can be seen, on Figure 2.16 (25-25 nodes) less SeDs are present than in Figure 2.15 (20-20 nodes). Thus, the MA is less loaded on the star graph obtained on 25-25 nodes, than on the star graph obtained on 20-20 nodes.



(a) *min-first*



(b) *max-first*

Figure 2.12: Comparison *min-first* and *max-first* with a star graph with the same SeD distribution.

Hierarchy shape

Figures 2.13, 2.14, 2.15 and 2.16 give an example of the shape of the hierarchy generated by respectively *min-first* on a 20-20 and 25-25 platform, and *max-first* on a 20-20 and 25-25 nodes platform.

Colored nodes are on the Sagittaire cluster, white nodes on the Capricorne cluster. “D” stands for `dgemm`, and “F” stands for Fibonacci.

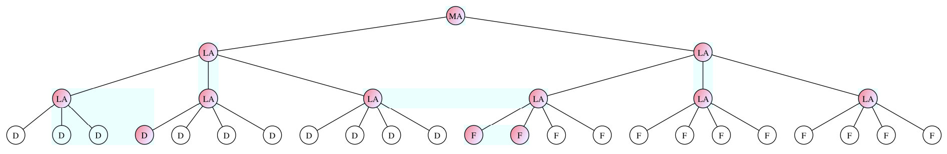


Figure 2.13: Hierarchy generated with *min-first* on 20-20 nodes.

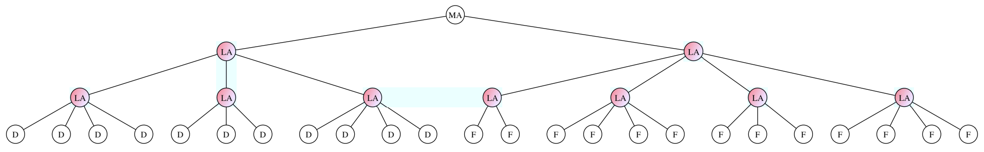


Figure 2.14: Hierarchy generated with *min-first* on 25-25 nodes.

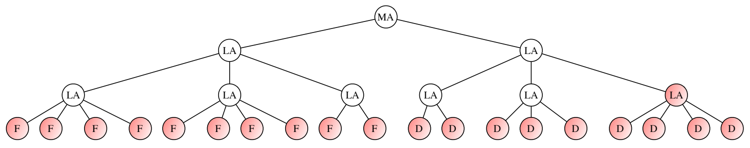


Figure 2.15: Hierarchy generated with *max-first* on 20-20 nodes.

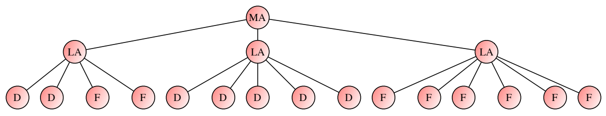


Figure 2.16: Hierarchy generated with *max-first* on 25-25 nodes.

2.5 Fully Heterogeneous

In this section we finally deal with a fully heterogeneous platform: each node j has a computing power of its own w_j, and the communication links between any two nodes j, j' are possibly all different: $B_{j,j'}$. Hence, the agents and servers models follow the general model presented in Section 2.2.

The problem when dealing with totally heterogeneous platforms is that we need information about both the location of the parent, and location of the children. Hence, the bottom-up approach we used so far will not be applicable, nor would be a top-down approach: we will not be able to build a level if we do not know the position of the parent in a bottom-up approach, and conversely we cannot build a level without knowing the position of the children in a top-down approach.

2.5.1 Genetic Algorithm Approach

As we cannot use an iterative approach to build a hierarchy without risking to have to test all possible solutions, we took a totally different approach: we rely on a *genetic algorithm* to generate a set of hierarchies, then evolve them, and finally select the best one among them.

In order to define our genetic algorithm, we need to describe a few notions: the objective function, crossover and mutations, and finally the evaluation strategy.

Objective function

The *objective function* has to encode all the goals we aim at optimizing in a hierarchy. It also needs to be subject to an order relation. Many genetic algorithm frameworks require that the objective function is encoded on an integer or floating point variable.

Our goal is the same as before: we aim at maximizing the minimum ρ_i/ρ^*, while minimizing the number of agents constituting the hierarchy. Hence our goal can be summarized with the following point of decreasing importance: (*i*) maximize $\min_{i\in\mathcal{R}} \{\rho_i/\rho^*\}$, (*ii*) then maximize the second ratio, the third…, (*iii*) minimize the number of agents to support the maximum throughput (*i.e.,* maximize $\mathcal{N}$ minus the number of agents). In order to encode all these points into a single value, we use the following encoding (presented in Figure 2.17): if we are given a precision ϵ on each ρ_i/ρ_i^* ratio, then we can encode them on $1/\epsilon$ digits; moreover, as the maximum number of agents is $\mathcal{N}-1$, we only need $\lceil\log_{10}(\mathcal{N})\rceil$ digits to encode the number of agents. Hence, on the whole we need $\mathcal{R}\times 1/\epsilon + \lceil\log_{10}(\mathcal{N})\rceil$ digits.

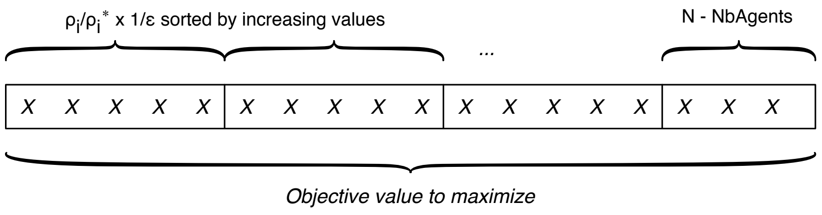

ρi/ρi* x 1/ε sorted by increasing values
N - NbAgents
...
X X X X X X X X X X X X X X X X X X
Objective value to maximize

Figure 2.17: Objective value encoding.

Actually, in order to guide a bit more the genetic algorithm towards convergence, we added two more metrics that we wish to minimize at the end of the fitness value: the number of agents that do not have any children (in order to remove really useless elements) and the depth of the hierarchy (this should not affect the throughput, but this impacts the response time of the hierarchy, and limits the formation of chains of agents).

Genotype

The *genotype* needs to encode the whole hierarchy: the parent/children relationship, and the type of each node (agent, server or unused). It can easily be encoded on two arrays of size $\mathcal{N}$: one for encoding the type of the nodes, and one for encoding the parent/children relationship. The *alleles*, *i.e.,* each value in the arrays, can have the following values. The *type* of a node can either be 0 if the node is unused, 1 if it is an agent, or $i \in \{2\ldots 2+\mathcal{R}\}$ if the node is a server of type $i-2$. The *parent* of a node i can either be itself if the node is unused (*i.e., type*$[i]=0$) or the MA, or the node number corresponding to the parent of i in the hierarchy. Genotypes are randomly generated when creating the first generation of individuals: nodes' types and relationship are randomly chosen in such a way that a valid hierarchy is created.

Crossover

We define a *crossover* between two hierarchies as follows. Crossovers are only made on the parent array. We randomly select two nodes (one on each hierarchy) and exchange the parent of both selected nodes. Figure 2.18 presents an example of crossover. Why not define a crossover which replaces a whole part of a hierarchy into another one? This approach works well for a small number of nodes, but it has a far too big impact on the hierarchy shape on a large number of nodes. As an example, consider hierarchies H_1 and H_2 in Figure 2.18, suppose a crossover between node 6 in H_1 and node 1 in H_2. Transferring node 6 into H_2 in place of node 1 would remove seven nodes, as node 6 is a server. Conversely, we cannot transfer node 1 in place of node 6 into H_1, as node 1 is the root of the hierarchy, and thus a transfer would lead to a loop in the hierarchy.

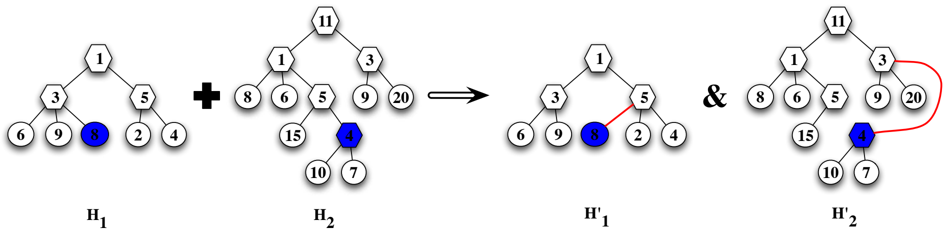


Figure 2.18: Crossover. Colored nodes are the one selected for crossover, within hierarchy elements represent the nodes' number.

Mutation

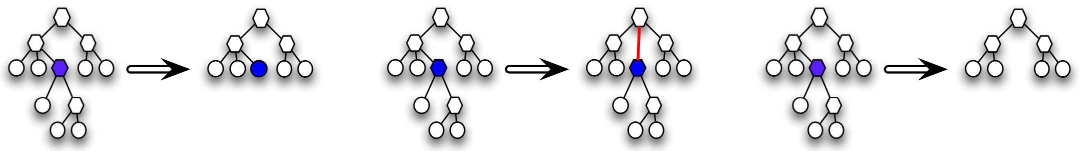
Mutations on a hierarchy can occur at different levels in the hierarchy. We define the following mutations, also presented in Figure 2.19:

– Hierarchy modification:

1. we randomly select a node to change its type. If the mutation changes the type from agent to unused or SED, or from SED to unused, then we remove the underlying hierarchy and modify the type of the node. If the type changes from unused to agent or SED, we randomly choose a parent among the available agents.

2. we randomly select a node that will choose a new parent among the available agents. We can end up with two hierarchies (if the new parent is the node itself). In this case we randomly select one of the two hierarchies, and delete the other.

– Pruning: a node is randomly selected, then its whole underlying hierarchy is deleted.



(a) Type mutation (b) Parent mutation (c) Pruning

Figure 2.19: Mutations. Colored node has been selected for mutation. Hexagons are agents, and circles are servers.

Hierarchy evaluation

In order to evaluate a hierarchy generated by our genetic algorithm, we first compute for all $i \in \mathcal{R}$ the throughput supported by the servers, we denote by $\rho_i^{\max}$ this throughput. Then, for each agent in the hierarchy, we compute the maximum throughput ρ_i for each service supported by the agent, we use $(\mathcal{LP}_5)$ to compute ρ_i.

$$\begin{array}{l}
\textbf{Maximize } \dfrac{\rho_1}{\rho_1^{\max}} \\
\textbf{Subject to} \\
\left\{\begin{array}{lll}
(1) & \forall i \in \mathcal{R} & 0 \le \rho_i \le \rho_i^{\max} \\
(2) & \forall i \in \mathcal{R}, i \neq 1 & \dfrac{\rho_1}{\rho_1^{\max}} = \dfrac{\rho_i}{\rho_i^{\max}} \\
(3) & & \displaystyle\sum_{i\in\mathcal{R}} \rho_i.\delta_i^j.\frac{w_{req_i}}{w_j} + \sum_{i\in\mathcal{R}} \rho_i.\left|Chld_i^j\right|.\frac{w_{resp_i}}{w_j} + \\
& & \displaystyle\sum_{i\in\mathcal{R}} \frac{\rho_i.\delta_i^j.m_{req_i}}{B_{j,f^j}} + \sum_{i\in\mathcal{R}}\sum_{k\in Chld_i^j} \frac{\rho_i.m_{resp_i}}{B_{j,k}} + \\
& & \displaystyle\sum_{i\in\mathcal{R}} \frac{\rho_i.\delta_i^j.m_{resp_i}}{B_{j,f^j}} + \sum_{i\in\mathcal{R}}\sum_{k\in Chld_i^j} \frac{\rho_i.m_{req_i}}{B_{j,k}} \le 1
\end{array}\right.
\end{array} \qquad (\mathcal{LP}_5)$$

Implementation and parameters

Genetic algorithms rely on quite a lot of different parameters. Each one of them can influence the quality of the result. Among them are the following:

- *Selection method*: when comparing x individuals, we need to choose which one should "survive." Different approaches exist: deterministic tournament, stochastic tournament, roulette or ranking. Our tests showed that the deterministic tournament was the one giving on average better results. Hence, we use this selection method in our experiments. When we force that the best parent replaces the weakest child if the child has a lower fitness, we talk about *weak elitism*. Weak elitism can possibly provide better solutions as it forces the algorithm to converge towards a *locally* good solution. However, it also reduces the population diversity, and thus, can lead to a local optimum. Our simulations tended to provide worst solutions when weak elitism was used, thus, in the following studies, we do not use it.
- *Size of the population*: with n machines, we varied the size of the population between $n/4$ and $2 \times n$. The worst results were always obtained for a size of $n/4$. The best results were obtained for populations having $5/4 \times n$ and $2 \times n$. On average, better results were obtained for populations of sizes between $5/4 \times n$ and $2 \times n$.
- *Probability of crossover and mutation*: crossover rate should generally be high, around 80%-90%, and mutation rate quite low, around 0.5%-1%. A very small mutation rate may lead to genetic drift, whereas a high one may lead to loss of good solutions. We chose set the mutation and crossover rates respectively to 1% and 80%.

We used the ParadisEO [56] framework to implement our genetic algorithm.

2.5.2 Quality of the Approach

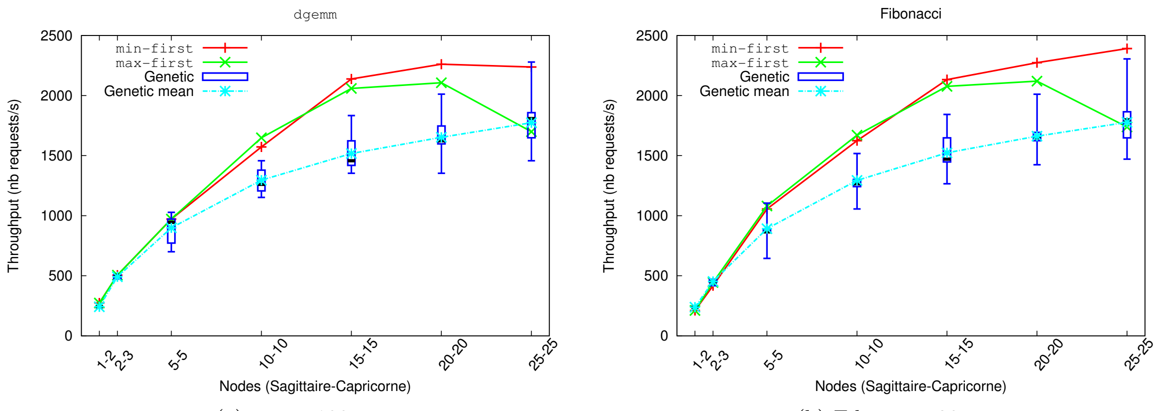
Genetic algorithms mainly depend on stochastic processes, thus we need to assess the quality of the results. As we do not have any other algorithm for fully heterogeneous platforms, we compare our genetic algorithm with the heuristics proposed in Section 2.4 for computation heterogeneous platforms.

Comparison with computation heterogeneous, communication homogeneous algorithm

We compare our Genetic Algorithm (GA) with *min-first* and *max-first*. GA was run on 1000 generations, on a population having at least 50 individuals (or $2 \times |V|$ if this is higher than 50). We used the deterministic tournament method of selection, and we ran 100 executions (100 seeds). Figure 2.20a and 2.20b respectively present the results for `dgemm` and Fibonacci services. As can be seen, even if on the mean GA do not obtain results as good as *min-first* or *max-first*, the best GA results closely follows the *min-first* heuristic. Table 2.6 presents the relative gain/loss obtained with the best solution of the genetic algorithm, compared to the *min-first* and *max-first* heuristics. As one can see, the loss is no bigger than 15%, and it gives better results than *max-first* for the larger platform. This confirms that our approach can be effective: even if one run of GA is not sufficient to obtain the best result, taking the best hierarchy over a few runs of GA can give us good results. Note that this is often the case with genetic algorithms, as it is an exploratory method. We are quite confident that the performance loss obtained with the GA solutions can be reduced by fine tuning the GA parameters.

Experiment		**Number of nodes**						
		1-2	**2-3**	**5-5**	**10-10**	**15-15**	**20-20**	**25-25**
min-first	`dgemm`	-11.3%	0.0%	5.7%	-7.3%	-14.3%	-11.0%	1.8%
	Fibonacci	12.8%	12.6%	4.6%	-6.6%	-13.6%	-11.6%	-3.6%
max-first	`dgemm`	-11.1%	0.0%	5.7%	-11.6%	-11.0%	-4.5%	34.1%
	Fibonacci	12.8%	5.9%	2.3%	-9.2%	-11.3%	-5.1%	32.6%

Table 2.6: Genetic algorithm gains/loss compared to the *min-first* and *max-first* heuristics. A positive number denotes a gain, whereas a negative one denotes a loss.



(a) `dgemm` 100 (b) Fibonacci 30

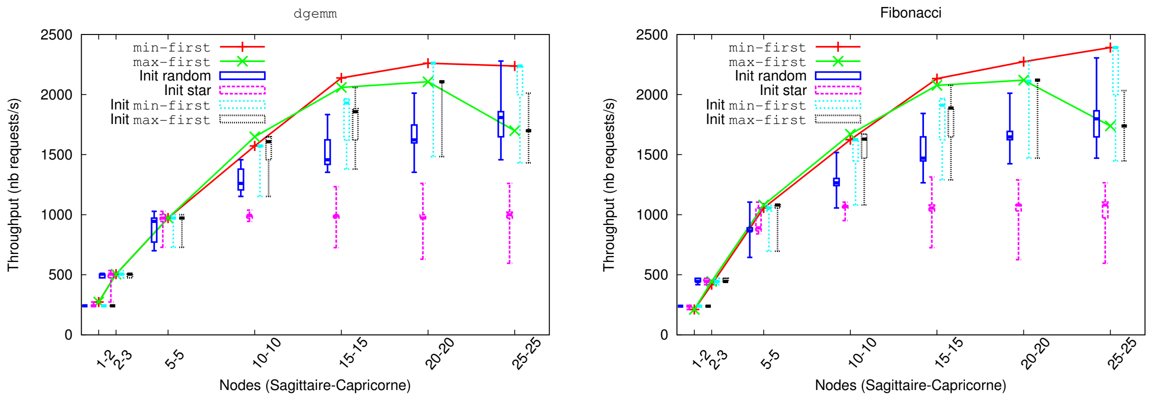
Figure 2.20: Comparison genetic algorithm results with *min-first* and *max-first* results.

Initialization difference

We also compared the results of the genetic algorithm for different initialization strategies. We compared four different strategies:

- *random*: we first randomly choose the number of nodes for each service, they are mapped on random nodes. Then, we create a random number of agents, and finally we just connect everything randomly, but in a way that creates a valid hierarchy. This is the initialization method used in the previous section.
- *min-first* or *max-first*: we use the *random* initialization for all but one hierarchy. We initialize this latter with the hierarchy found by *min-first* or *max-first*.
- *star graph*: we randomly choose the type of each nodes, and ensure that only one can be an agent. Then elements are connected as a star graph under the agent.

Figures 2.21a and 2.21b present the results. The star graph gives the worst results, as new levels of hierarchy can be created only through mutations, and they do not occur often. Without much surprise, the *min-first* and *max-first* initialization give the best results as they are guided by the result of the bottom-up algorithm. Finally, the random initialization gives results that are in between *min-first* and *max-first* on the mean, but whose best result is almost as good as *min-first* method. These results also assess the effectiveness of our approach, as starting from totally random hierarchies, we obtain results as good as with the *min-first* heuristic.



(a) `dgemm` 100 (b) Fibonacci 30

Figure 2.21: Different initialization methods.

2.5.3 Experiments

We ran experiments on DIET hierarchies, with the same services as before: `dgemm` 100 and Fibonacci 30. We used three clusters present on three different sites of Grid'5000: Sagittaire in Lyon, Chti in Lille and Paradent in Rennes. Their respective computing power is $w_{sagittaire} = 3249MFlops$, $w_{Chti} = 3784MFlops$ and $w_{Paradent} = 4378MFlops$. Figure 2.22 present the comparison between theoretical and experimental throughput, and Table 2.7 presents the relative error. As can be seen, the experiments follow the model. Even though the error can be quite big for small platforms (less than 10 nodes), the performance prediction becomes more accurate for larger platforms, as the relative error remains lower than 16%. The higher errors obtained for small platforms can be explained by the fact that the platform benchmarks have been done by stressing the network and machines. This stress is more easily attained when many nodes are involved in the experiments.

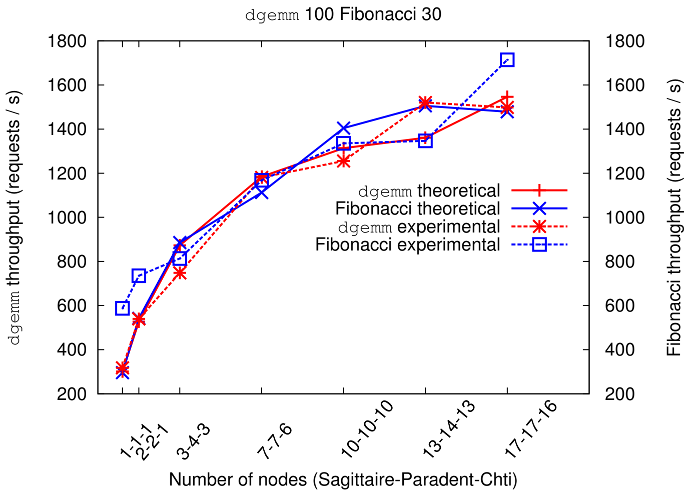


Figure 2.22: `dgemm` 100, Fibonacci 30 theoretical and experimental throughput, obtained with GA.

Client	**Sagittaire-Paradent-Chti**						
	1-1-1	**2-2-1**	**3-4-3**	**7-7-6**	**10-10-10**	**13-14-13**	**17-17-16**
`dgemm`	5.42%	2.29%	14.26%	0.33%	4.44%	11.73%	3.12%
Fibonacci	98.63%	35.70%	8.16%	4.96%	4.91%	10.53%	15.95%

Table 2.7: Relative Error, using genetic algorithm.

Figure 2.23 presents the shape of the hierarchy obtained for a 17-17-16 platform. As can be seen, not all the available nodes are used: as previously, adding more nodes does not necessarily provide better performances as it tends to overload the agents. Blue nodes are on the Chti cluster, red ones on Paradent, and the white ones on Sagittaire.

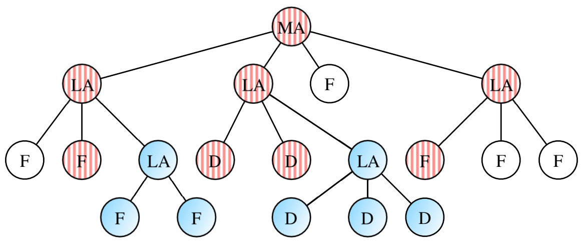


Figure 2.23: Hierarchy shape for a 17-17-16 platform.

2.6 Elements Influencing the Throughput

It is not straightforward to obtain the best throughput out of a given hierarchy. Several parameters have to be taken into account in order to obtain the best throughput. We give here a list of some of the problems we encountered while running our experiments and the solutions we used to overcome these problems. During the first experiments we made, the results did not fit exactly the model. Analyzing the execution traces lead us to four main parameters having a great impact on the performance.

2.6.1 Bad Scheduling

The basic scheduling implemented within DIET relies on the time of the last request a SED has solved. When a request is sent down the hierarchy, each SED replies the *time since last solve*, *i.e.,* the time elapsed since the SED has solved a request. Hence, when multiple requests are sent in the hierarchy at the same time, a SED can reply the same value of *time since last solve* for multiple requests. Thus the same SED is likely to be chosen for a set of requests. This, of course, may overload some SED and do not give any work to the others. Hence, in order to cope with this, we need to activate an option of DIET which allows a client to make its own scheduling. The client keeps a local list of available SED for the service, and choose the server in a round robin fashion (which is what is implied by our model, as all servers are meant to have the same amount of load). Activating this option allowed us to increase the throughput up to more than 80% for some deployed hierarchies, using exactly the same number of clients. Note that bypassing the DIET schedulers by using local caching might lead to problems on a highly dynamic platform. In our case we used a stable and well controlled platform, hence the effectiveness of the clients' scheduling.

Also note that this does not discard our model, as our model supposes that the load of the jobs are distributed on all servers according to their computing power. Currently DIET is not able to schedule correctly requests arriving at a high throughput.

2.6.2 Logging

Turning on or off the logging in DIET can greatly influences the obtained throughput. Indeed, several log messages are sent through CORBA by each element of the hierarchy whenever a request is sent and solved. When logging is turned on, we observe periods of burst and periods of starvation on the throughput. There can be a difference of several hundreds of requests per seconds between two periods, and the starvation periods tend to dominate the whole execution. This leads, on a 60 seconds run, to a 30% - 40% throughput loss when logging is enabled. This is contradictory to what has been shown in [52], however, in our case the tests we run have a number of requests per seconds an order of magnitude higher than what the authors present, hence the bigger impact. Thus, using the logging system when DIET deals with a high throughput of requests worsen the performances.

2.6.3 Number of Clients

The number of clients involved in the system also influences the throughput. The first point is of course the fact that too few clients will not send enough requests in the system to fully load it. Conversely, too many clients will overload the system as no queuing of the requests is done at the SED level. Hence, in order to find the best throughput in our experiments, we needed to find the correct number of clients. One more point has to be taken into account. Having lots of clients, each sending a request at a time, is more *expensive* than having less clients, but threaded ones that are able to send several requests in parallel. We compared the execution obtained on the same hierarchy when first launching 150 clients on 40 nodes for each `dgemm` and Fibonacci services, and secondly with 10 `dgemm` clients with 15 threads each, and 2 Fibonacci clients with 75 threads. Each thread can send a single request in the system at a given time (*i.e.,* a new request can only be sent when the previous one has been solved). Hence, the two configurations are equivalent in terms of number of "clients", *i.e.,* the number of requests that can be sent to the hierarchy at a given time. What we observed is that with threaded clients the throughput is higher. We gained between 10% and 31% on the throughput for each service with this method.

Hence, during our experiments, we ran at least 10 different combinations number of clients/number of threads for each service. We kept only the best combination for each experiment. Of course, on a production platform, the number of clients depends on the users' needs, and thus this parameter cannot be adjusted.

2.6.4 OmniORB Configuration

OmniORB has by default some limits on the number of threads CORBA clients and servers can use at the same time. This can limit the throughput of the platform, especially if "large" data transfers have to take place (this is the case for example with `dgemm` 500). Thus, in order to improve the throughput, we need to increase the number of threads clients and servers can use. This is done respectively with `maxGIOPConnectionPerServer` (default value: 5), *i.e.,* the maximum number of concurrent connections a CORBA client will open to a single server, and `maxServerThreadPoolSize` (default value: 100), *i.e.,* the maximum number of threads a CORBA server is allowed to allocate. We set those values respectively to 100 and 1000. With these values, we gained around 17% - 18% on the number of requests per seconds on experiments with larger data transfer (*e.g.,* `dgemm` 500). They require that the connection between the client and the server lasts longer, hence limiting the number of concurrent calls that can be done to a given SED.

We applied all the above-mentioned optimizations during the experiments presented in this chapter. Clients made their own Round-Robin, logging was disabled, several number of clients/number of threads were tested, and omniORB parameters where tuned so as to obtain the best results.

2.7 Conclusion

In this chapter, we have presented a model for hierarchical middleware performance, as well as three algorithms for automatically finding a relevant hierarchy on different platforms. Compared to previous work, our approach allows the deployment of several services in the same hierarchy. We also deal with fully heterogeneous platforms. Two of our algorithms are based on a bottom-up approach, and rely on a linear program to build the levels of agents. The last algorithm is based on a genetic algorithm, in order to cope with the complexity of totally heterogeneous platforms. We conducted experiments with a hierarchical middleware, Diet, on the French experimental platform, Grid'5000. The results validate the model accuracy, and show the effectiveness of our approach compared to classical deployment approaches. Finally, the experiments allowed to detect several points that are likely to impact the performance of the middleware, and thus, we were able to improve Diet performances.

Part II

Organizing Nodes

Chapter 3

Preliminaries on Clustering

Deploying applications on a distributed platform can be improved by bringing a new "line of sight" on the platform. Usually described as a graph with no particular high level structures, a platform can be enhanced by creating clusters of nodes, according to a given metric. Clustering provides a new level of abstraction, and offers a better insight of the platform performances. In this chapter, we give related work on distributed graph clustering, as well as an introduction to the concept of self-stabilization. The first section is dedicated to general centralized graph clustering. Then, in Section 3.2 we present some distributed graph clustering algorithm. We then move on to the definition of self-stabilization in Section 3.3, before presenting related work on self-stabilizing clustering algorithms, and more specifically self-stabilizing k-clustering algorithms in Section 3.4.

3.1 Graph Clustering

Many problems can be represented by a graph, be it in numerical simulations such as in cosmology, automobile industry, or in networking problems. Graph partitioning is a commonly used technique to either solve problems, or at least reduce their complexity. In this section, we first present general graph clustering methods and libraries, then we present solutions specific to network clustering.

3.1.1 General Clustering Techniques

The main clustering libraries share the same global techniques for designing their algorithms. One of the main approach is the multilevel partitioning, which first coarsen the graph before creating an initial partition which is then projected on the initial graph, and refined in the uncoarsening phase. They all provide *bipartitioning* algorithms based on derived versions of Kernighan-Lin method [113]: the graph is divided into two parts of almost equal weight, while minimizing the communication cost between the two parts, *i.e.,* minimizing the cost of the cut. Another method, called the *spectral approach*, relies on the determination of eigenvectors from the Laplacian matrix of the graph. Here are three of the main clustering libraries.

Zoltan [26] is developed at the Sandia National Laboratories, Livermore, USA. It provides a suite of parallel partitioning algorithms and data migration tools to dynamically redistribute data.

MeTiS [15] is developed at the Karypis Laboratory, Minnesota, USA. It provides three libraries: MeTiS, PARMeTiS and h-MeTiS which are respectively version for sequential clustering, parallel clustering, and hypergraph and circuit clustering. MeTiS provides multi-level partitioning techniques, as well as an adaptive mesh refinement framework.

Scotch [139] is developed at the LABRI Laboratory, Bordeaux, France. It proposes many recursive bipartitioning algorithms, mapping algorithms and also a multilevel framework. A version for parallel clustering is also provided: PT-Scotch . It offers partition refinement techniques relying on either a genetic algorithm, or on a diffusion based algorithm.

3.1.2 Network Clustering Techniques

Special methods have been proposed in the case where the graph represents a network of machines. Network clustering is particularly important for example to provide efficient routing algorithms, or to guaranty a quality of service, or to help scheduling tasks, or again to collect statistics on clients connections on a server. Anyhow, network clustering is always related to a notion of performance that one wants to improve or guaranty. For example, on large scale networks due to resource consumption (energy, and bandwidth), a hierarchical structure is better than using a "flat" one [115].

IP clustering is used in [117] in order to determine where requests arriving to a web server came from, they base their algorithm on the determination of proxies and spiders. Extracting the IP addresses as well as the network masks and grouping the nodes by longest common prefix have been studied in [158], and seems to provide good results. This approach also provides a way to obtain a representation of the network topology [118]. Some clustering algorithm are based on Euclidean spaces. For example, [168] is based on GNP [130] (Global Network Positioning) to determine nodes coordinates within an n dimensional space, clusters are then created based on latencies. A commonly used technique is to elect leaders, around which clusters are formed. Typically, the creation of dominating sets is used [104, 167], *i.e.,* a subset D of nodes is selected, and all nodes in the graph either belong to D or are connected by an edge to at least one of the nodes of D. Spectral analysis can also be used, for example to analyze Internet topology [100].

The algorithms presented in this section are *centralized* algorithms, as one machine needs to know everything about the platform. However, centralized algorithms work well for as long as we can easily have a global view of the platform. This implies having a global knowledge, which is not always possible especially in distributed environments such as grids. Hence, the natural step to cope with this problem, is to distribute the problem to have processes working jointly to reach a common solution.

3.2 Distributed Clustering

Many distributed clustering algorithms exist in the literature. A clustering provides a hierarchical organization of a network, which consists in grouping nodes into clusters, and, in many clustering algorithms, electing a *clusterhead* that is in charge of managing the cluster. A clusterhead can be in charge of performing performance measurements, or enhancing the spatial reuse of time slots and codes... Moreover, clustering reduces the amount of information stored and maintained in nodes such as routing or topology information; and decrease the transmission overhead. Changes made within a cluster can also be kept locally, and thus will not "pollute" the whole network.

Solutions for *ad hoc networks* aim at grouping nodes which are "geographically close" in the network, given a certain criterion. For example, the Lowest-ID Clustering (LID) [87] algorithm, is based on the nodes identifier (ID) in order to build clusters within which all nodes are one hop away from an elected leader (also called clusterhead) within the cluster. Least Cluster Head Change (LCC) [67] tries to limit clusterheads changes when there are modifications in the network. High-Connectivity Clustering [98] is based on the nodes degree instead of their ID. The algorithms Distributed Clustering Algorithm (DCA), and Distributed and Mobility-Adaptive Clustering (DMAC) presented in [40] use a weight assigned to nodes to choose the clusterheads. [37] propose a distributed algorithm which organizes the nodes hierarchically. Clustering can also be used to reduce the power consumption of the nodes, an energy aware clustering algorithm is presented in [99]. Many other distributed clustering methods can be found in [138].

The above mentioned algorithms, albeit working on distributed networks, aren't able to cope with errors, crashes that are inherent properties of distributed systems. In the next section, we present an approach to deal with any kind of faults occurring in a distributed system.

3.3 Concept of Self-Stabilization

We now introduce the concept of *Self-Stabilization*, a powerful technique to design fault-tolerant systems.

3.3.1 Self-Stabilization

The concept of *Self-Stabilization* has been introduced in 1974 by Dijkstra in [77]. It is a general concept for designing fault-tolerant systems. Whatever the error arising in a self-stabilizing system, this latter will be able to recover, and after a phase of stabilization, will be able to return to an intended behavior in a finite time. Hence, once an error has occurred, regardless of the state in which the system is left, *i.e.,* regardless of the states of the processes, and the messages in the communication links, the system will be able to recover from any kind of errors, be it the loss of some nodes, or the corruption of some variables. Thus, the self-stabilizing property claims that there is no need for a proper reset of the variables, nor for a static network, as a self-stabilizing algorithm is guaranteed to recover *automatically* from an arbitrary initialization in a finite time. This property contrasts with the typical fault-tolerance concept that only states that under all state transitions, the system will never deviate from a correct state. Conversely, a non self-stabilizing system driven to an illegitimate configuration by some perturbations may not return to a correct behavior.

3.3.2 Distributed System Assumptions

Consider a connected undirected network, represented by a graph $G = (V, E)$, where V is the set of nodes (Vertices), and E is the set of links (Edges) between nodes. Note that contrary to the previous chapter, we do not restrict the graph to fully connected platforms. Distributed algorithms are modeled as distributed state machines performing a sequence of *steps*. For a given node (process), a step consists in first reading its local and direct neighborhood's *state*, and then, depending on those states' values, performing an action. A process is said to be *enabled* if it can perform at least one action. Determination of the neighborhood's state is done through communications. Two main communication models exist: *message passing* and *shared memory*. A commonly used model is Dijkstra's theoretical model [77], more commonly referred to as the *shared memory* model, where a process can directly read the variables of its neighbors in one atomic operation. In this model each node can directly read it's neighbors variables, and read and write it's own. The grain of *atomicity* may also vary [79]. In the *composite atomicity* model, a process can read the state of the system and execute an action (*i.e.,* modify its own variables) in one atomic step. On the contrary, in the *read/write atomicity* model, a process can read the state of the system in one atomic step, and then execute an action in one atomic step, hence the configuration may change between the read and write steps.

As we are evolving in a distributed system, we need to *schedule* the different processes steps. We assume that each transition from a configuration to another is driven by a *scheduler*, also called a *daemon*. A daemon can be *synchronous* if at each step all processes are activated and execute at the same time, *distributed* if it can activate simultaneously any subset of processes, or *fully distributed* if it is distributed and that all activated processors read their input registers, perform local computations, but may delay the writing into their registers after the following daemon activation has occurred. Moreover, a processor may not be chosen again by the fully distributed daemon until it has written its output registers. Daemons also have different levels of *fairness*. A *fair* daemon ensures that if a process P is continuously enabled, then P is continuously selected by the daemon. A *weakly fair* daemon ensures that if a process P is continuously enabled, then P must eventually be selected by the daemon. Finally, an *unfair* daemon might never choose a continuously enabled process P, unless P is the only enabled process.

One should always keep in mind, that a self-stabilizing algorithm can only stabilize if the time between two faults is *long enough* for the algorithm to converge. No convergence can be proven if some failures constantly appear. That is why we say that stabilizing will *eventually* happen, after the *final* fault in the system.

3.4 The k-clustering Problem and Self-Stabilizing Clustering

The *k-clustering problem* [76] consists in partitioning a network into clusters, within which all nodes are no farther than a distance k from the clusterhead in the cluster. Thus, the result of a k-clustering are subgraphs with a diameter at most $2k$, k being a given data of the problem. The diameter of a graph being the greatest distance between any pair of vertices. We can distinguish several forms of this problem: clusters can be overlapping or disjoint, and the network can be represented by an unweighted or weighted graph. In the next chapter, we deal with the weighted version of the problem, with disjoint clusters.

The purpose of such a clustering is to provide a local view of the platform, and help routing within the clusters. Typically, clusterheads can be in charge of collecting information about the machines states in the cluster and constructing an overview of its cluster state. It can also provide routing tables, so as to transmit messages in less than $2k$ between any two nodes in the cluster. Clustering facilitates the reuse of resources, which improves the system capacity.

To the best of our knowledge, no previous work dealt with weighted graphs. Amis *et al.* [32] give the first distributed solution to the k-clustering problem on unweighted graph. The time and space complexities of their solution are $O(k)$ and $O(k \log n)$, respectively. Spohn and Garcia-Luna-Aceves [150] give a distributed solution to a more generalized version of the k-clustering problem. In their algorithm, a parameter m is given, and each process must be a member of m different k-clusters. The k-clustering problem discussed in this dissertation is then the case $m = 1$. The time and space complexities of the distributed algorithm in [150] are not given. Fernandess and Malkhi [89] give an algorithm for the k-clustering problem that uses $O(\log n)$ memory per process, takes $O(n)$ steps, provided a *breadth first search* (BFS) tree for the network is already given – a BFS tree has a designated *root*, and from each node, the path from that node through the BFS tree to the root is the shortest possible path in the network.

The first *self-stabilizing* solution to the k-clustering problem was given by Datta *et al.* in [72]; this solution takes $O(k)$ rounds and $O(k \log n)$ space. Another self-stabilizing solution was proposed in [70]; this algorithm needs $O(n)$ rounds but only $O(\log n)$ space. Both solutions use the hop metric, and are thus unable to deal with more general weighted graphs.

Many self-stabilizing algorithms have been proposed in the literature for constructing clusters in distributed network. In [42], Bein *et al.* give a link-cluster algorithm where each node can be at a distance two of any clusterhead. Dolev and Tzachar present in [81] a stochastic self-stabilizing clustering algorithm. In [169], Zu *et al.* give an algorithm for constructing a minimal dominating set.

Other self-stabilizing clustering algorithms deal with weighted graphs, where weights are placed on the vertices, not on the edges. For example, Johnen and Nguyen give in [111] an algorithm to partition the network into 1-hop clusters, *i.e.,* the algorithm computes a *dominating set*, a set S such that ever node is a neighbor of some member of S. This article presents self-stabilizing versions of DMAC [40] and GDMAC [39]. The authors also give a robust version of both algorithms in [112], *i.e.,* after one round the network is partitioned into clusters, and stays partitioned during construction of the final clusters.

A self-stabilizing algorithm for clusters formation under a *density* criterion is presented in [128] by Mitton *et al.*. The density criterion (defined in [127]) is used to select clusterheads – a node v is elected a clusterhead if it has the highest density in its neighborhood, and the cluster headed by v contains all nodes at distance less or equal to two from v.

Algorithms presented in this section can only deal with unweighted graphs, *i.e.,* only the number of hops between nodes is taken into account. This approach may be relevant when dealing with ad-hoc networks, but may hide the true communication cost. Indeed, when communication costs are not homogeneous in a platform, taking into account only the number of hops in no longer sufficient. For example, consider the latency within a network, it can vary a lot depending if you communicate with a neighbor accessible through a local network, or through a wide area network. Thus, in the next chapter, we present a self-stabilizing distributed k-clustering algorithm for weighted graphs.

Chapter 4

Dynamic Platform Clustering

In the previous part of the dissertation, we introduced how to efficiently deploy a hierarchical middleware on a fully connected platform. However, real large scale platforms, in general, cannot be represented by a fully connected network. Most networks have to be represented by a graph where connections occur only between well identified nodes. Running an application without taking into account the underlying network can lead to dramatic loss of performances. Hence the need to run applications only on group of well connected nodes arises. But then, what if the platform is modified? What if errors or crashes occur in the system? Grids are not reliable platforms, they are dynamic and error prone environments. Thus, taking into account locality in the network is an improvement for obtaining good performances, but this notion of locality can be modified throughout time due to dynamicity.

In this chapter [1], we tackle the problem of grouping nodes on a platform according to a given distance k. So far, we studied the problem of obtaining a good throughput on the requests served by a middleware. What if now we want to address the problem of having a good response time on the submission part, *i.e.,* the time between the instant a client sends a request, and the instant it receives a list of available servers? Due to latencies between nodes, the time to search servers can be quite long if we do not deploy the middleware on well connected nodes. Hence, a notion of *locality* arises. Two problems are addressed in this chapter:

1. **k-clustering.** Given an additive weighted metric on a network (*e.g.,* the latency between nodes), we aim at grouping the nodes into sets of non-empty connected subgraphs, within which each node is at a distance at most k from a particular node elected in this subgraph. Hence, within each subgraph, two nodes are no further apart than $2k$. Going back to middleware deployment, suppose that a limit δ on the response time to obtain a list of servers is given. We only need to fix $k = \delta/2$, find a k-clustering on the network, and deploy the middleware within a subgraph to ensure the response time. To the best of our knowledge, we present the first solution to the k-clustering problem on weighted graphs.
2. **Self-stabilization.** Grids being dynamic and error prone platforms, we aim at designing a clustering algorithm that can cope with such dynamicity, and recover from any kind of errors that could happen during the execution of an algorithm. Any modification or error in the system can be taken into account. Our approach ensures that *eventually* our k-clustering algorithm will reach a correct state.

The rest of the chapter is organized as follows. In Section 4.1, we give the preliminary concepts on self-stabilization. In Section 4.2, we describe more formally the addressed problem. As our solution to the k-clustering problem makes use of several self-stabilizing algorithms (also called modules), we present new results on combining self-stabilizing algorithms under an unfair daemon in Section 4.3. Sections 4.4 to 4.6 are then dedicated to presenting the different modules of our algorithm. Finally, before concluding, we present complexity results in Section 4.7 and simulation results in Section 4.8.

1. The work presented in this chapter has been published in journal [CDDL10], international conference [CDDL09] and national conference [Dep09].

4.1 System Assumptions

We now present more formally the concepts of self-stabilization, which has been introduced in the previous chapter. The *state* of a process is defined by the values of its variables. A *configuration* of the network is a product of the state of all processes. If γ is the current configuration, then $\gamma(P)$ is the current state of each process P. An *execution* of an algorithm, $\mathcal{A}$ is a sequence of states $e = \gamma_0 \mapsto \gamma_1 \mapsto \ldots \mapsto \gamma_i \ldots$, where $\gamma_i \mapsto \gamma_{i+1}$ means that it is possible for the network to change from configuration γ_i to configuration γ_{i+1} in one step. We say that an execution is *maximal* if it is infinite, or if it ends at a *sink*, *i.e.,* a configuration from which no further execution is possible.

The *program* of each process consists of a set of variables and a finite set of actions, each protected by a *guard*. The *guard* of an action in the program of a process P is a Boolean expression involving the variables of P and of its neighbors. The *statement* of an action of P updates one or more variables of P. An action can be executed only if it is *enabled*, *i.e.,* its guard evaluates to true. A process is said to be *enabled* if at least one of its actions is enabled. A step $\gamma_i \mapsto \gamma_{i+1}$ consists of one or more *enabled* processes executing an action. We consider that any enabled node P is *neutralized* in the computation step $\gamma_i \mapsto \gamma_{i+1}$, if P is enabled in γ_i and not enabled in γ_{i+1}, but does not execute any action between these two configurations. This situation could occur if some neighbors of P change some of their registers in such a way as to cause the guards of all actions of P to become false.

We assume that each transition from a configuration to another is driven by a *scheduler*, also called a *daemon*, and we consider this latter to be *unfair and distributed* (see Chapter 3). The *unfairness* means that even if a process P is continuously enabled, P might never be selected by the daemon, unless, at some step, P is the only enabled process. The *distributed* property means that at a given step, if one or more processes are enabled, the daemon selects an arbitrary non-empty set of enabled processes to execute an action.

In order to compute the time complexity, we use the notion of *round* [80], which captures the speed of the slowest process in an execution. We say that a finite execution $\varrho = \gamma_i \mapsto \gamma_{i+1} \mapsto \ldots \mapsto \gamma_j$ is a *round* if the following two conditions hold:

1. Every process P that is enabled at γ_i either executes or becomes neutralized during some step of ϱ.
2. The execution $\gamma_i \mapsto \ldots \mapsto \gamma_{j-1}$ does not satisfy condition 1.

We define the *round complexity* of an execution to be the number of disjoint rounds in the execution, possibly plus one more if there are some steps left over.

Finally, we say that an algorithm $\mathcal{A}$ is *self-stabilizing* if, starting from a completely arbitrary configuration, the network will eventually reach a legitimate configuration. More formally, we assume that we are given a *legitimacy predicate* $\mathcal{L}_\mathcal{A}$ on configurations. Let $\mathbb{L}_\mathcal{A}$ be the set of all *legitimate* configurations, *i.e.,* configurations which satisfy $\mathcal{L}_\mathcal{A}$: *correct* configurations. Then we define $\mathcal{A}$ to be *self-stabilizing* to $\mathbb{L}_\mathcal{A}$, or simply *self-stabilizing* if $\mathbb{L}_\mathcal{A}$ is understood, if the following two conditions hold:

1. **Convergence.** Every maximal execution contains some member of $\mathbb{L}_\mathcal{A}$.
2. **Closure.** If an execution e begins at a member of $\mathbb{L}_\mathcal{A}$, then all configurations of e are members of $\mathbb{L}_\mathcal{A}$.

We say that $\mathcal{A}$ is *silent* if every execution is finite. In other words, starting from an arbitrary configuration, the network will eventually reach a *sink*, *i.e.,* a configuration where no process is enabled.

4.2 The k-Clustering Problem

We now formally define the problem solved in this chapter. Let $G = (V, E)$ be a connected graph (network) consisting of n nodes (processes), with positively weighted edges. We suppose that the network contains at least two nodes, $n \geq 2$. For any $x, y \in V$, let $w(x, y)$ be the *distance* from x to y, defined to be the least weight of any path from x to y. We will assume that the edge weights are positive integers. We also define the radius of a graph G as follows:

$$radius(G) = \min_{x \in V} \max_{y \in V} \{w(x, y)\}$$

Given a positive integer $k \geq 1$, we define a *k-cluster* of G to be a non-empty connected subgraph of G of radius at most k, *i.e.,* such that all processes in the cluster are within distance k of a designated leader process, called the *clusterhead*.

We define a *k-clustering* of G to be a partitioning of V into k-clusters. The *k-clustering problem* is then the problem of finding a k-clustering of a given graph. [2] We also require that a k-clustering specifies one node, which we call the *clusterhead* within each cluster, which is within k of all nodes of the cluster, and a *shortest path tree* rooted at the clusterhead which spans all the nodes of the cluster.

A set of nodes $D \subseteq V$ is a *k-dominating set*[3] of G if, for every $x \in V$, there exists $y \in D$ such that $w(x, y) \leq k$. A k-dominating set determines a k-clustering in a simple way; for each $x \in V$, let $Clusterhead(x) \in D$ be the member of D that is closest to x. Ties can be broken by any method, such as by using IDs. For each $y \in D$, $C_y = \{x : Clusterhead(x) = y\}$ is a k-cluster, and $\{C_y\}_{y \in D}$ is a k-clustering of G.

We say that a k-dominating set D is *optimal* if no k-dominating set of G has fewer elements than D. The problem of finding an optimal k-dominating set, or equivalently, a k-clustering with the minimum possible number of clusters, is known to be $\mathcal{NP}$-hard [32]. Our algorithm attempts to find a k-clustering which has "few" clusters in a reasonable time.

4.3 Unfair Composition of Self-Stabilizing Algorithms

Before presenting our solution for dealing with the k-clustering problem, we first present the conditions under which self-stabilizing algorithms can be combined to form a new self-stabilizing algorithm, which works under the unfair daemon. The underlying question behind this work is: "can we reuse different self-stabilizing algorithms, combine them and obtain a self-stabilizing algorithm?" By combination, we mean that given two algorithms A_1 and A_2, the stabilized behavior of A_1 is used as input by A_2. In [79] are presented the conditions for *fair combination* of self-stabilizing algorithm. Fair combination denotes the fact that each process executes a step of each algorithm infinitely often. In this section, we consider the problem of combining distributed algorithms under an unfair daemon. This problem is not entirely trivial; for example, what we have discovered is that a naive combination of self-stabilizing algorithms might not be self-stabilizing.

We define a *partial execution* of a distributed algorithm A to be a sequence of configurations such that, other than the first one, each follows from its predecessor by one or more processes executing an action of A.

We define an *execution* of A to be a partial execution which is either infinite or ends at a configuration where no process is enabled.

1. We say that an execution is *unfair* if it is infinite and if there is some process that executes only finitely many times and is continuously enabled from some point on.
2. We say that an execution is *weakly fair* if it is not unfair. We say that A is *weakly fair* if every execution of A is weakly fair.
3. We say that an execution is *strongly fair* if it is either finite, or there is no process that executes only finitely many times. We say that A is *strongly fair* if every execution of A is strongly fair.

Note: a strongly fair execution is also weakly fair, and a strongly fair algorithm is also weakly fair.

Lemma 4.1. *Every distributed algorithm has a weakly fair execution.*

Proof. At each step, select the set of all enabled processes. ■

The logical question that arises is: "Is there a strongly fair execution for every distributed algorithm?" The answer is unfortunately "No."

2. There are several alternative definitions of k-clustering, or the k-clustering problem, in the literature.
3. Note that this definition of the *k-dominating set* is different than another well known problem consisting in finding a subset $V' \subseteq V$ such that $|V'| \leq k$, and such that $\forall v \in V - V', \exists y \in V' : (x, y) \in E$. [97]

4.3.1 The Daemon

The scheduler (daemon) chooses an execution of the algorithm A.
– We say that a daemon is *weakly fair* if it always chooses a weakly fair execution.
– We say that the daemon is *unfair* if it can choose any execution.

We say that a distributed algorithm A *works under the weakly fair daemon* if every weakly fair execution of A has whatever properties are required in the problem specification.

We say that a distributed algorithm A *works under the unfair daemon* if every execution of A has whatever properties are required in the problem specification.

Lemma 4.2. *If A is a weakly fair distributed algorithm, then A works under the unfair daemon if and only if A works under the weakly fair daemon.*

4.3.2 Input Variables

Normally, input variables are never discussed. However, in most cases, a distributed algorithm has variables that never change their values. We can call these *input variables.* In the literature, it seems to always be assumed that the input variables are constant during the execution of an algorithm.

But what if we want to combine algorithms, so that the input variables of the second module (algorithm) are computed by the first module? In this case we cannot consider input variables to be constants in every cases.

1. An *input variable* of an algorithm A is a variable that is used by A but is never changed by A. We call the vector of all input values of all nodes the *input configuration.*
2. The usual definition of an algorithm S being *self-stabilizing* is that, given that the input configuration is correct and never changes, the network will eventually be in a legitimate configuration.

   Of course, each problem specification gives a definition of what it means for an input configuration to be correct, and what it means for a configuration to be legitimate. We will assume that if the configuration is legitimate, the input configuration is correct.

   (a) *S is self-stabilizing under the unfair daemon* if every execution where the input configuration is correct and never changes is eventually in a legitimate configuration.

   (b) *S is self-stabilizing under the weakly fair daemon* if every weakly fair execution where the input configuration is correct and never changes is eventually in a legitimate configuration.

Whether an algorithm A is weakly or strongly fair depends on the definition of a configuration of the algorithm. The normal definition of configuration allows any process to have any values for its variables, but the values of its constants are uniquely specified. But what about input variables? Since this issue is not normally even discussed in the literature, we need to clarify it for our purposes.

We will adopt the definition that an algorithm A can have constants, whose values are given in the problem specification, and could also have input variables, which could take on a range of values, but whose values cannot be changed by A (if we never combine algorithms, the distinction between these is moot). We then define a partial execution of A to be a sequence of configurations where the input variables can be initialized to have any values in their range, and where during each step one or more processes execute an action of A. Thus, the input variables are unchanged throughout the entire sequence.

We summarize the classification of the variables of an algorithm A as follows:

1. *Constants,* which have values given in the problem specification.
2. *Input variables.* Each input variable has a range of possible values. There is a defined set of input configurations call *correct* input configurations. Input variables cannot be changed by A.
3. *Local variables.* Variables which can be changed by A, and are used only for the internal computations of A.
4. *Output variables.* Variables which can be changed by A, and which are intended to be read by some agent or algorithm outside A.

We also require that, if the configuration is legitimate, then no output variable can change. We say that the output variables are *stable.*

For example, in the third module of K-CLUSTERING, which we fully describe in Section 4.6.1, k and $P.id$ are constant, $P.parent$, $P.leader$, $P.level$, and $P.minid$ are input variables, $P.dist$ and $P.span$ are local variables, and $P.maxminid$ is an output variable.

4.3.3 Combining Algorithms

Suppose that A and B are strongly fair self-stabilizing algorithms on the same network. That means that each process P has all the variables of both A and B. We classify those variables as follows:

1. Constants.
2. Input variables of A that are not visible to B. These cannot be changed.
3. Variables that are input variables of both A and B. These cannot be changed.
4. Input variables of B which are not visible to A. These cannot be changed.
5. Local variables of A.
6. Local variables of B.
7. Output variables of A which are input variables of B. These can be changed by A but not by B.
8. Output variables of A which are not input variables of B.
9. Output variables of B.

Finally, the combination algorithm *Combine*(A, B) is defined as follows (also see Figure 4.1):

1. The input variables of *Combine*(A, B) are defined to be the variables of classes 2, 3, and 4 above.
2. The output variables of *Combine*(A, B) are defined to be the variables of classes 8, and 9 above, possibly together with some variables of class 7.

P is enabled to execute an action of *Combine*(A, B) if and only if P is either enabled to execute an action of A or enabled to execute an action of B. If P executes an action of *Combine*(A, B), then P executes both an action of A and an action of B, if both are enabled, otherwise, it executes one or the other.

Correctness and legitimacy for the three algorithms, A, B, and *Combine*(A, B), must be defined to satisfy the following conditions:

1. If the input configuration of *Combine*(A, B) is correct, then the input configuration of A is correct.
2. If the input configuration of *Combine*(A, B) is correct and the configuration of A is legitimate, then the input configuration of B is correct.
3. A configuration of *Combine*(A, B) is legitimate if and only if the configurations of both A and B are legitimate.

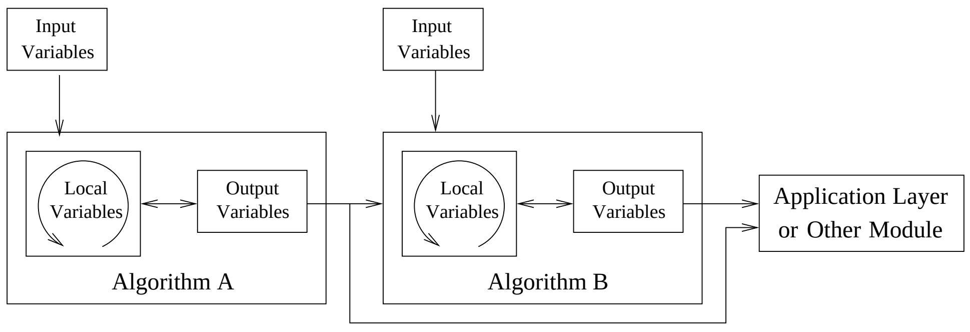


Figure 4.1: *Combine*(A, B)

Lemma 4.3. *If both A and B are strongly fair and self-stabilizing, then Combine(A, B) is strongly fair and self-stabilizing.*

Proof. We first prove that *Combine*(A, B) is *strongly fair*, *i.e.,* that every execution of *Combine*(A, B) is either finite, or that no process executes only finitely many times.

Let $E = \gamma_0, \gamma_1, \ldots$ be an execution of *Combine*(A, B). We write $\gamma_i = (\alpha_i, \beta_i)$, where α_i is a configuration of A, and β_i is a configuration of B. Let $E_A = \alpha_0, \alpha_1, \ldots$ which might not be a partial execution of A because consecutive configurations could be equal. Let E'_A be the partial execution of A obtained from E_A by eliminating configurations which are the same as their predecessors.

Case I: E is finite. In this case, we are done.

Case II: E is infinite, and E'_A is infinite. Then, since A is strongly fair, every process executes infinitely many actions of A in E'_A, and hence infinitely many actions of *Combine*(A, B) in E.

Case III: E is infinite, and E'_A is finite. Pick T such that A does not execute beyond the T^{th} step of E, that is, $\alpha_i = \alpha_T$ for all $i > T$. Then $E_B = \beta_T, \beta_{T+1}, \ldots$ is an execution of B. Since B is strongly fair, each process executes infinitely many actions of B during E_B, and thus infinitely many actions of *Combine*(A, B) during E.

Thus, in any case, every execution of *Combine*(A, B) is either finite, or no process executes only finitely many times, *i.e., Combine*(A, B) is strongly fair.

We now prove that *Combine*(A, B) is *self-stabilizing*, *i.e.,* that every execution where the input configuration is correct and never changes is eventually in a legitimate configuration. We use the same notations as above.

Assume that the input configuration of *Combine*(A, B) is correct at γ_0. Then, the input configuration of A is correct at α_0. We claim that E'_A is an execution of A.

Case I: E is finite. Then, at the last configuration of E, no process is enabled to execute an action of *Combine*(A, B), and hence no process is enabled to execute an action of A. Thus, E'_A is an execution of A.

Case II: E is infinite, and E'_A is infinite. Then E'_A is an execution of A.

Case III: E is infinite, and E'_A is finite. Suppose that E'_A is not an execution of A. Then, at α_T, there is some process P which is enabled to execute an action of A, but that process is never selected during the remainder of the sequence E. This contradicts the fact that E contains infinitely many actions of every process.

This completes the proof of the claim that E'_A is an execution of A. It follows that α_i is eventually a legitimate configuration of A.

This finally leads us to the end of the proof of Lemma 4.3, *i.e.,* that *Combine*(A, B) is self-stabilizing.

Assume that the input configuration is correct. Since E'_A is an execution of A, there is some T such that α_T is a legitimate configuration of A. Hence, for any $i \geq T$, the output variables of A have legitimate values, and are stable. Thus, for any $i \geq T$, the input variables of B are correct and are the same as at γ_T. Let E'_B be the sequence of configurations of B, starting at β_T, with duplicates removed. Then E'_B is an execution of B, and thus will eventually be in a legitimate configuration of B at which the output variables of B are stable.

In conclusion, eventually both A and B will be in legitimate configurations, and the output variables of both will be stable, and we are done. ■

We have the necessary conditions to build self-stabilizing algorithms out of several self-stabilizing modules, under the unfair daemon. We will now iteratively build our solution for the k-clustering problem. We first present a non self-stabilizing algorithm for solving the *Best Reachable* problem, which is at the core of our approach to solve the k-clustering problem. Then, we modify it so as to make it self-stabilizing. Finally, we present two other self-stabilizing modules, and combine them to form our self-stabilizing solution to the k-clustering problem.

4.4 *Best Reachable* Problem

We now go back to a non self-stabilizing algorithm for solving what we call the *Best Reachable* problem. It aims at finding a *best* value within a weighted network under a distance constraint.

We define the *Best Reachable* problem on a network as follows. We are given a positive weight function w on edges, and we let $w(P, Q)$ be the minimum weight of any path from P to Q, as before. We are also given a number k, the *allowed distance*. Without loss of generality, the weight of any edge is at most $k + 1$.

Each process P has a *value* $P.\Theta$, of some type, and each process must calculate the *best* value of $Q.\Theta$ over all processes Q within that allowed distance of P. More specifically, each P must calculate $best\{Q.\Theta : w(P, Q) \leq k\}$.

Best means maximum under any given ordering. In our code, we will write "$\succ$" for a given order relation on values of Θ, and we say that $P.\Theta$ is *best* if $P.\Theta \succeq Q.\Theta$ for all processes Q. Throughout, we write $\mathcal{N}_P$ for the set of all neighbors of P.

We first present in Section 4.4.1 the algorithm, and then its proof in Section 4.4.2.

4.4.1 Algorithm NSSBR

We now give a distributed algorithm, NSSBR, presented in Algorithm 4.1, for the best reachable problem. Each process P has variables $P.best$, whose value of the best value of Θ that P has found so far, $P.dist$, the distance from P of the nearest Q for which $Q.\Theta = P.best$, and $P.span$, whose meaning is as follows: $P.best = best\{Q.\Theta : w(P, Q) < P.span\}$. That is, P has so far found the best value of Θ among all process which are closer than $P.span$, but not among those whose distance from P is greater than or equal to $P.span$.

Initially, $P.best = P.\Theta$ and $P.dist = 0$, because P only considers of its own value of Θ. The initial value of $P.span$ is the shortest distance from P to any neighbor, since P has not searched any neighbor for a better value.

As the algorithm proceeds, each process P repeatedly iterates the main loop, shown as lines 1.4 through 1.12 in the code below. The loop will iterate until $P.span > k$, which will indicate that P has searched all processes of distance at most k to find the best value of Θ.

The only way that P can become aware of values of Θ beyond its immediate neighborhood is through its neighbors. For example, if X is within k of P and $X.\Theta$ is the best value of Θ within k of P, then P must have a neighbor Y which is on the shortest path from P to X, and P will learn about $X.\Theta$ from Y. At some point in the computation, $Y.best = X.\Theta$, and P will update $P.best$ to that value.

However, there is a complication. Even though P learns about $X.\Theta$ via Y, it could be that there is some better value of Θ within k of Y, but not within k of P. This means that $Y.best$ will eventually be better than $X.\Theta$. We must make sure that P can read $Y.best$ before that happens.

Algorithm 4.1 NSSBR: A Non-Self-Stabilizing Algorithm for Best Reachable

```
1.1:  $P.best \leftarrow P.\Theta$
1.2:  $P.dist \leftarrow 0$
1.3:  $P.span \leftarrow \min\{w(P,Q) : Q \in \mathcal{N}_P\}$
1.4:  while $P.span \leq k$ do
1.5:    if $\forall Q \in \mathcal{N}_P : ((Q.best \succeq P.best) \vee (P.dist + w(P,Q) > k)) \wedge (w(P,Q) + Q.span > P.span)$ then
1.6:      if $\exists Q \in \mathcal{N}_P : Q.best \succ P.best$ and $Q.dist + w(P,Q) = P.span$ then
1.7:        $P.best \leftarrow \max_{\succ}\{Q.best : Q \in \mathcal{N}_P \text{ and } Q.dist + w(P,Q) = P.span\}$
1.8:        $P.dist \leftarrow P.span$
1.9:      end if
1.10:     $P.span \leftarrow \min \begin{cases} \min\{X.span + w(P,X) : X \in \mathcal{N}_P\} \\ \min\{X.dist + w(P,X) : X \in \mathcal{N}_P \text{ and } X.best \succ P.best\} \end{cases}$
1.11:   end if
1.12: end while
```

Each process P has the code presented in Algorithm 4.1. In Line 1.7, "$\max_{\succ}$" denotes maximum with respect to the order relation "$\succ$."

In order to fit Algorithm 4.1 into our model of computation, we assume that each P executes lines 1.1 through 1.3 of the code immediately, *i.e.,* before any other process reads its values. Lines 1.4 through 1.12 are executed as one atomic step, so that a neighbor of P cannot, for example, read the new value of $P.best$ until the new values of $P.dist$ and $P.span$ are also computed.

The code of Algorithm 4.1 is not self-stabilizing. We will later show how to modify it to make it self-stabilizing.

4.4.2 Proof of Correctness for NSSBR

In this section, we prove that Algorithm 4.1 converges and that after convergence, for all process P, $P.best = \max_{\succ}\{Q.\Theta : w(Q,P) \le k\}$.

Intuition. As Algorithm 4.1 proceeds, each process P tries to find the best value of Θ within an increasing distance. It keeps track of the *search radius*, $P.span$, as well as $P.best$, the best value of Θ within that distance of itself. It also keeps track of $P.dist$, which is the distance to the process whose Θ value is $P.best$ (in case more than on such process exists, $P.dist$ is the smallest choice of distance).

Loop invariant. We now define the *loop invariant* of the main loop of Algorithm 4.1, which is the conjunction of the following invariants, each of which holds for all choices of processes P, X, and Y.

LI(i)(P): $0 \le P.dist \le k$ and $P.dist < P.span$

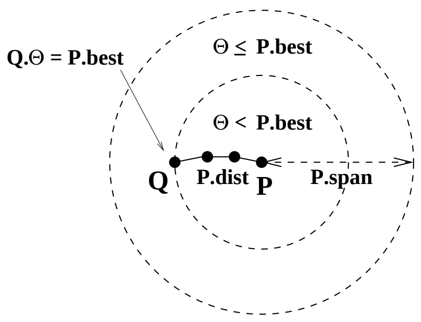
LI(ii)(P): $P.best = \max_{\succ}\{X.\Theta : w(P,X) < P.span \text{ and } w(P,X) \le k\}$

LI(iii)(P): $P.dist = \min\{w(P,X) : X.\Theta = P.best\}$

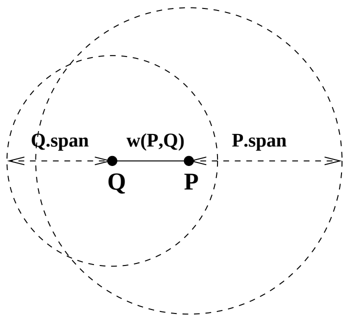
LI(iv)(P,X): $P.span \le X.span + w(P,X)$ if $X \in \mathcal{N}_P$

LI(v)(P,X,Y): If $Y \in \mathcal{N}_P$, $w(P,X) < P.dist$, and $w(P,X) + w(P,Y) \le k$, then $X.\Theta \preceq Y.best$

Explanation of the loop invariant. We now explain the intuition behind the loop invariant. Figure 4.2a illustrates LI(i), LI(ii), and LI(iii). For each process P, the distance from P to the nearest process Q such that $Q.\Theta = P.best$ is stored as $P.dist$, and no process closer to P has a better value of Θ. Furthermore, P has determined that no better Θ exists among all processes closer than $P.span$.



(a) Invariants LI(i), LI(ii), and LI(iii).



(b) Invariant LI(iv).

Figure 4.2: Invariants LI(i), LI(ii), LI(iii), and LI(iv).

Figure 4.2b illustrates LI(iv). If Q is a neighbor of P, then $Q.span + w(P,Q) \ge P.span$. The basic reason for this invariant is that P derives all information about other processes from its neighbors.

By far the hardest invariant to explain is LI(v). Suppose $Y \in \mathcal{N}_P$, $w(P,X) < P.dist$, and $w(P,X) + w(P,Y) \le k$. Pick processes U and V such that $U.\Theta = Y.best$ and $V.\Theta = P.best$, as illustrated in Figure 4.3. Suppose also that $w(Y,V) > k$. Thus, it could happen that $X.\Theta$ is the largest value of Θ within k of Y.

The only way that Y can know about $X.\Theta$ is through its neighbor, P. But $P.best = V.\Theta$, which is larger than $X.\Theta$, and thus $P.best$ will never again be equal to $X.\Theta$.

To avoid error, we must ensure that $X.\Theta$ will not be needed by Y in the remaining part of the computation. The invariant LI(v), which states that $Y.best \succeq X.\Theta$, guarantees this.

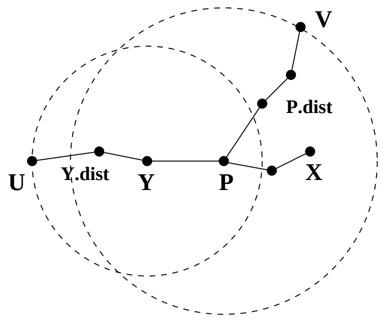


Figure 4.3: Invariant LI(v): $U.\Theta = Y.best$, $V.\Theta = P.best$, $w(U,Y) = Y.dist$, $w(P,V) = V.\Theta$, $w(P,X) < P.dist$, and $w(Y,X) \le k < w(Y,V)$.

Figure 4.4 gives an example of how a calculation can go wrong if LI(v) is not used. In that figure, $D.best$ will be unable to achieve its correct value of $3 = B.\Theta$, since $C.best$ has already found a better value, namely $4 = A.\Theta$, for $k = 2$.

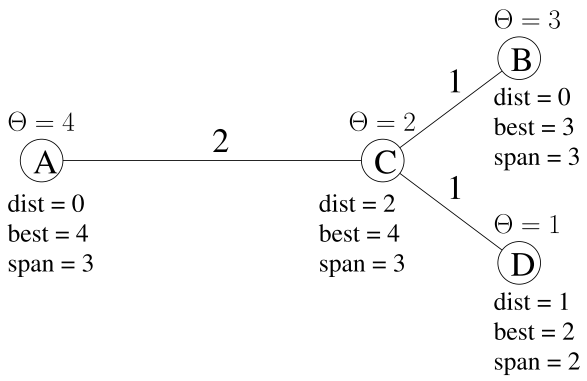


Figure 4.4: Example showing the necessity of LI(v). In the figure, $k = 2$, and the invariant LI(v)(C, D, B) is false, although all other parts of the loop invariant hold. It is impossible for $D.best$ to achieve its correct value of 3.

We will now prove that NSSBR solves the Best Reachable Problem. We first show that the loop invariant holds once Lines 1.1 through 1.3 are executed, and that it holds thereafter. Then, we show that at least one process modifies its variables for as long as we do not have $P.span > k$ for all process P. This leads to the conclusion that NSSBR solves the Best Reachable Problem.

Lemma 4.4. *The loop invariant holds after each process executes Lines 1.1 through 1.3 of the code of Algorithm 4.1.*

Proof. Recall our assumption that no process iterates the main loop of Algorithm 4.1 until after all processes have *initialized*, *i.e.,* have executed Lines 1.1 through 1.3. After all processes have initialized, then $P.span = \min\{w(P,Q) : Q \in \mathcal{N}_P\} > 0$, $P.dist = 0$, and $P.best = P.\Theta$ for all P. The invariants LI(i) through LI(iv) are then trivially true, while LI(v) holds vacuously. ■

Lemma 4.5. *If the loop invariant holds before a step, then it holds after that step.*

Proof. Assume that the loop invariant holds before a given step. During the step, some subset of processes executes the loop of Algorithm 4.1. For each process P, let $P.best$, $P.dist$, and $P.span$ be the values of P's variables before the step, and let $P.best'$, $P.dist'$, and $P.span'$ be the values after that step.

We will also write LI(i), LI(ii), *etc.* for the invariants before the step, and LI(i)$'$, LI(ii)$'$, *etc.* for the invariants after the step.

For our proof, we fix a process P, and assume that LI(i)(P), LI(ii)(P), and LI(iii)(P) hold, and that LI(iv)(P, X) and LI(v)(P, X, Y) hold for all processes X and Y. We then prove that the corresponding "primed" invariants, LI(i)$'(P)$, LI(ii)$'(P)$, *etc.* hold.

We will consider three cases, depending on the execution of P during the step.
Case I is where the condition of the **if** statement on Line 1.5 is false for P. In this case, P does not change its variables during the step.
Case II is where that condition is true, but the condition of the **if** statement on Line 1.6 is false for P. In this case, P executes Line 1.10, but does not execute Lines 1.7 or 1.8.
Case III is where the conditions on Lines 1.5 and 1.6 are both true. In this case, P executes Lines 1.7, 1.8, and 1.10.
In Case III, we will choose $Q \in \mathcal{N}_P$ such that $P.span = Q.dist + w(P, Q)$, and $Q.best$ is maximum subject to that condition; thus $Q.best \succ P.best$. We will also choose a process R such that $w(Q, R) = Q.dist$ and $R.\Theta = Q.best$. In Cases I and II, Q and R are undefined.

We first show that in the three cases, the variables $P.span$ and $P.dist$ can only increase or keep the same value, and that the variable $P.best$ can only become better (with regards to $\succ$) or keep the same value. This leads to the proof that in Case I, Lemma 4.5 holds, and that only Cases II and III need further investigation.

Claim A: In Case III, $P.best' \succ P.best$ and $P.span' > P.span = P.dist' > P.dist$.

Proof (of Claim A):
$P.dist' = P.span > P.dist$ by LI(i)(P), and $P.best' = Q.best \succ P.best$ by execution of Line 1.7.
We need only show that $P.span' > P.span$. Suppose not. Then, either $\exists X \in \mathcal{N}_P$ such that $X.span + w(P, X) < P.span$, which contradicts LI(iv)(P, X), or $\exists X \in \mathcal{N}_P$ such that $X.best \succ P.best$ and $X.dist + w(P, X) < P.span$. But, by the choice of Q and by LI(ii)(P), $Q.best \succeq X.best$ for all $X \in \mathcal{N}_P$ such that $X.dist + w(P, X) \leq P.span$, which leads to a contradiction. The last case is when $\exists X \in \mathcal{N}_P$ such that $X.span + w(P, Q) \leq P.span$, but this case is prohibited by condition Line 1.5. □

Claim B: If $X \in \mathcal{N}_P$ and $P.best \prec X.best$, then $P.span \leq w(P, X) + X.dist$, *i.e.,* if $P.best$ is worst than $X.best$, then it means that P as not searched as far as X has.

Proof (of Claim B):
By LI(iii), we can pick Y such that $Y.\Theta = X.best$ and $w(X, Y) = X.dist$. By LI(ii) and by the triangle inequality, $P.span \leq w(P, Y) \leq w(P, X) + w(X, Y) = w(P, X) + X.dist$. □

Claim C: For any process P, $P.best' \succeq P.best$, $P.dist' \geq P.dist$, and $P.span' \geq P.span$.

Proof (of Claim C):
In Case I, there is nothing to prove, as P does not change its variables. In Case III, we are done by Claim A. Consider Case II. Trivially, $P.best' = P.best$ and $P.dist' = P.dist$, as Lines 1.7 and 1.8 are not executed, since condition Line 1.6 is false in this case. $X.span + w(P, X) \geq P.span$ for all $X \in \mathcal{N}_P$, by LI(iv)(P, X), and $X.dist + w(P, X) \geq P.span$ for all $X \in \mathcal{N}_P$ such that $X.best \succ P.best = P.best'$, by Claim B. Thus, $P.span' \geq P.span$. □

In Case I, LI(i)$'(P)$, LI(ii)$'(P)$, LI(iii)$'(P)$, and LI(iv)$'(P, X)$ hold trivially, since LI(i)(P), LI(ii)(P), LI(iii)(P), and LI(iv)(P, X) hold and the variables of P do not change. LI(v)$'(P, X, Y)$ holds in the three cases, since LI(v)(P, X, Y) holds, and since $Y.best' \succeq Y.best$, by Claim C applied to Y. This completes the proof of the lemma in Case I, and thus henceforth, we assume that we have either Case II or Case III.

We now give five new claims required to prove the remaining invariants. The first two claims, Claims D and E, are related to the proof that LI(ii)$'(P)$, LI(iii)$'(P)$ and LI(iv)$'(P, X)$ hold.

Claim D: $P.span' \leq \min \begin{cases} \min \{X.span + w(P, X) : X \in \mathcal{N}_P\} \\ \min \{X.dist + w(P, X) : X \in \mathcal{N}_P \text{ and } X.best \succ P.best\} \end{cases}$ for any process P.

Proof (of Claim D):
Since P executes Line 1.10 during the step, that execution makes the claim true. □

Claim E: For any process X, if $w(P, X) < P.span'$ and $w(P, X) = k$, then $X.\Theta \preceq P.best'$.

Proof (of Claim E):
In Case II, $P.span' = P.span$ and $P.best' = P.best$, and we are done by LI(ii)(P). Consider Case III. Choose $Y \in \mathcal{N}_P$ such that $w(P, X) = w(P, Y) + w(Y, X)$. Then $Y.best \succeq X.best$ and $Y.span + w(P, Y) \geq P.span' > w(P, X)$ by Claim D.

Suppose $Y.best \preceq P.best'$. Then $w(X, Y) = w(P, X) - w(P, Y) < Y.span$, and thus $X.\Theta \preceq Y.best \preceq P.best'$, by LI(ii)$(Y)$.

On the other hand, suppose $Y.best \succ P.best$. Then $w(P, X) < P.span' \leq Y.dist + w(P, Y)$, and hence $w(Y, X) < Y.dist$. We also have that $w(Y, X) + w(Y, P) = w(P, X) \leq k$. By LI(v)$(Y, X, P)$, we have $X.\Theta \preceq P.best \prec P.best'$. □

Finally, the last three claims, Claims F, G and H, are related to the proof that LI(iii)$'(P)$ holds.

Claim F: In Case III, $P.dist' = w(P, R) = w(P, Q) + w(Q, R)$.

Proof (of Claim F):
By the triangle inequality and LI(iv)(P, R),
$w(P, R) \leq w(P, Q) + w(Q, R) = w(P, Q) + Q.dist = P.span \leq w(P, R)$, and $P.dist' = P.span$. □

Claim G: There is some process X such that $w(P, X) = P.dist'$.

Proof (of Claim G):
In Case II, $P.dist' = P.dist$, and we are done by LI(iii)(P). In Case III, let $X = R$. We are done by Claim F. □

Claim H: For any process X:
(a) If $w(P, X) < P.dist'$, then $X.\Theta \prec P.best'$.
(b) If $w(P, X) = P.dist'$, then $X.\Theta \preceq P.best'$.

Proof (of Claim H):
In Case II, $P.dist' = P.dist$ and $P.best' = P.best$, and we are done by LI(iii)(P). Consider Case III. Choose $Y \in \mathcal{N}_P$ such that $w(P, X) = w(P, Y) + w(Y, X)$. Then $Y.best \succeq P.best$ and

$$Y.span > P.span - w(P, Y) = P.dist' - w(P, Y) \geq w(P, X) - w(P, Y) = w(Y, X)$$

since the condition in Line 1.5 holds, and thus $X.\Theta \preceq Y.best$, by LI(ii)(Y).

Sub-case (i): $Y.best = P.best$. Then $X.\Theta \leq Y.best = P.best \prec P.best'$, and thus both (a) and (b) hold.

Sub-case (ii): $Y.best \succ P.best$.

(a): By LI(v)(Y, X, P), $X.\Theta \preceq P.best \prec P.best'$.
(b): $Y.dist \geq P.span - w(P, Y) = w(Y, X)$, by Claim B. If $Y.dist = w(Y, X)$, then $Y.best \preceq Q.best$ by our choice of Q, and thus $X.\Theta \preceq Y.best \preceq Q.best = P.best'$. If $Y.dist < w(Y, X)$, then $X.\Theta \preceq P.best \prec P.best'$ by LI(v)(Y, P, X). □

Armed with those five new claims, we can now finish the proof of the lemma in Cases II and III.

We first show that LI(i)$'(P)$ holds. In Case II, $P.span' \geq P.span$ by Claim C. Since LI(i)(P) holds before the step, we have $0 \leq P.dist = P.dist' \leq k$ and $P.dist' = P.dist < P.span \leq P.span'$. In Case III, we have $0 < P.dist' < P.span'$, by Claim A. Since the loop condition in Line 1.4 holds before the step, $k \geq P.span = P.dist'$.

LI(ii)$'(P)$ follows from Claim E, and the fact that $w(P, R) < P.span'$ and $R.\Theta = P.best'$ in Case III.

LI(iii)$'(P)$ follows from Claims G and H.

We now show that LI(iv)$'(P, X)$ holds. Assume $X \in \mathcal{N}_P$. $X.dist' \geq X.dist$ and $X.span' \geq X.span$, since Claim C holds for X. By Claim D, we are done.

The case of LI(v)$'(P, X, Y)$ has already been taken care of: we already proved that it held in the three cases.

Hence, we showed that the loop invariant holds after a step, if it held before the step, which completes the proof of Lemma 4.5. ■

Lemma 4.6. *If there is at least one process whose value of span is at most k, then there is at least one process P such that P can iterate the loop of Algorithm 4.1, and such that during that iteration, at least one variable of P changes.*

Proof. Let $\mathcal{P} = \{P : P.span \leq k\}$. Pick $P \in \mathcal{P}$ such that $P.best$ is minimum. If there is more than one choice, pick P such that $P.span$ is minimum. We will show that P changes at least one of its variables during its next iteration of the loop.

We first claim that P satisfies the condition of the **if** statement in Line 1.5. Suppose not. Then, there is some $Q \in \mathcal{N}_P$ such that $P.dist + w(P, Q) \leq k$ and $Q.best \prec P.best$, or $w(P, Q) + Q.span \leq P.span$.

Suppose $Q.span + w(P, Q) \leq P.span$. By the minimality of $P.best$, we have $Q.best \succeq P.best$. If $Q.best = P.best$, then $w(P, Q) + Q.span \leq P.span$, by the choice of Q, which contradicts the minimality of $P.span$ in our choice of P. Thus $Q.best \succ P.best$. Pick R such that $R.\Theta = Q.best$ and $w(Q, R) = Q.dist$. By LI(i)(Q) and the triangle inequality, we have:

$$w(P, R) \leq w(P, Q) + w(Q, R) \leq w(P, Q) + Q.dist < w(P, Q) + Q.span \leq P.span$$

Since $R.\Theta \succ P.best$, this contradicts LI(ii)(P).

Otherwise, $Q.best \prec P.best$, and thus $Q.span > k$. Pick a process R such that $R.\Theta = P.best$ and $w(R, P) = P.dist$. Then, $w(R, Q) \leq w(R, P) + w(P, Q) = P.dist + w(P, Q) \leq k < Q.span$ and $R.\Theta = P.best \succ Q.best$, which contradicts LI(ii)(Q). This proves the claim that P satisfies the condition in Line 1.5.

We need to show that P changes at least one variable during the resulting iteration. There are two cases. Case I: There is some $Q \in \mathcal{N}_P$ such that $Q.best \succ P.best$ and $Q.dist + w(P, Q) = P.span$. In case of a tie, pick that Q which has the maximum value of $Q.best$. Then P will execute Lines 1.7 and 1.8, changing $P.best$ to $Q.best$, and increasing both $P.dist$ and $P.span$.

Case II: Not Case I. Then P will not execute Lines 1.7 and 1.8, but will execute Line 1.10. We need to show that $P.span$ will increase. Let X be any neighbor of P. If $X.best \succ P.best$, then, since Case I does not hold, and by LI(i), $P.span < X.dist + w(P, X) < X.span + w(P, X)$. Otherwise, by the choice of P,

$X.span \geq P.span$, and thus $P.span < X.span + w(P, X)$. It follows that $P.span$ increases when Line 1.10 is executed. ■

Theorem 4.1. *Algorithm 4.1 solves the Best Reachable Problem.*

Proof. By Lemmas 4.4, 4.5, and 4.6, the loop invariant of Algorithm 4.1 holds at all times, and the algorithm will continue to execute as long as the loop condition, in Line 1.4 of the code, remains true for at least one process.

We need only show that the algorithm cannot keep changing variables forever. Whenever a process P changes any of its variables, the values of the changed variables increase, by Claim C in the proof of Lemma 4.5. There are at most n possible values of $P.best$. Since $P.dist$ is always equal to $w(P, Q)$ for some process Q, there are at most n possible values of $P.dist$. Since $P.span$ is always either equal to $w(P, Q)$ for some $Q \neq P$, or is greater than k, it also can take on at most n different values during the execution. Thus, Algorithm 4.1 converges.

Upon convergence $P.span > k$ for all P, and by LI(ii)(P), the value of $P.best$ is correct. ■

4.5 Self-Stabilizing Best Reachable

In this section, we give a self-stabilizing algorithm, SSBR (presented in Algorithm 4.2), for the Best Reachable problem. SSBR makes use of NSSBR as a module, and also requires the construction of a rooted breadth first search (BFS) tree. SSBR is a wave algorithm using broadcast and convergecast waves [138, 159], the mean of the BFS tree is to support those waves. We use the composite atomicity model of computation [77, 78]. Each process can read its own registers and those of its neighbors, but can only write to its own registers. The evaluations of the guard and executions of the statement of any action is presumed to take place in one atomic step. The algorithm is given in Section 4.5.1, followed by its proof in Section 4.5.2. An example of execution is given in Section 4.5.3.

4.5.1 Algorithm SSBR

We will use SSBR, also denoted as Algorithm 4.2, as a module as defined in Section 4.3. For that reason, it will be explicitly designed with input and output parameters, much like a subroutine in a program.

We assume that every process has an identifier (ID), $P.id$, which is given, and does not change, and that IDs are unique.

The inputs of Algorithm 4.2, SSBR, include outputs of some self-stabilizing algorithm which elects a leader and constructs a BFS tree rooted at that leader. We will refer to this algorithm as SSLEBFS. The outputs of SSLEBFS are $P.parent$, the ID of the current parent of P in SSLEBFS, $P.leader$, of ID type, but possibly not the ID of any process in the network, and $P.level \geq 0$, an integer. When SSLEBFS has converged, *i.e.,* reached a legitimate configuration, $P.leader$ is the ID of the root process, $P.level$ is the length of the shortest path from P to the root (in number of hops), and $P.parent$ is the parent of P in the BFS tree; the parent of the root is itself.

A process P does not execute any action of SSBR if it detects that the BFS tree is incorrect. The following conditions must hold for each P if the BFS tree is correct.

1. If $Q \in \mathcal{N}_P$, then $Q.leader = P.leader$.
2. $P.level = 0$ if and only if $P.leader = P.id$.
3. If $P.level = 0$ and $Q \in \mathcal{N}_P$, then $Q.parent = P$.
4. If $Q \in \mathcal{N}_P$, then $|Q.level - P.level| \leq 1$.
5. If $Q \in \mathcal{N}_P$ and $P.parent = Q$, then $P.level = Q.level + 1$.

We say that P is *locally correct* if the above conditions hold.

Lemma 4.7. *The BFS tree is correct if and only if P is locally correct for all P.*

In addition, we assume a function Θ on processes, whose value we refer to as $P.\Theta$ for each process P, and a specified ordering of the values of Θ. In the code for Algorithm 4.2, we refer to that ordering using the symbol "$\succ$."

The variables which are under the control of SSBR are as follows:

- $P.status \in \{working, finished, resting, ready\}$. We will say that P is working, is finished, is resting, or is ready.
- $P.stable\_best$, the output of SSBR.
- All the variables of NSSBR, namely $P.best$, $P.dist$, and $P.span$.

The execution of SSBR consists of two parts: *status correction* and *normal execution.* During normal execution, which presumes that the BFS tree is correct, four different *status waves* are alternately broadcast and convergecast, as shown in Figure 4.5. During each complete cycle of waves the values of $P.best$ is recomputed, and is compared to $P.stable\_best$, the output variable of SSBR. $P.stable\_best$ is then updated, if necessary, to agree with $P.best$. Between those updates, $P.stable\_best$ does not change; thus, eventually, $P.stable\_best$ is stable.

We say that *$P.status$ is incompatible with $P.parent.status$* if the current combination of status values of those two processes cannot occur during the normal part of the execution of SSBR. During status correction, $P.status \leftarrow P.parent.status$ if the values are incompatible. Figure 4.6 shows the eight combinations of incompatible values.

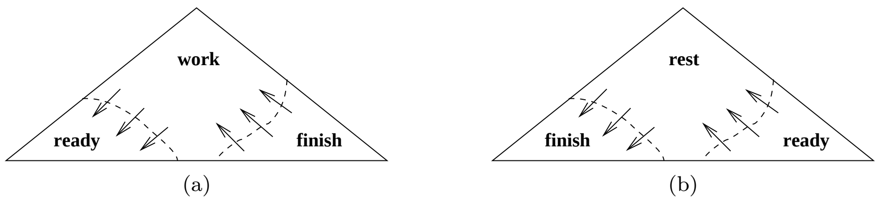


Figure 4.5: Broadcast waves *working* and *resting* and convergecast waves *finished* and *ready*. The *finished* wave could start before the *working* wave is completed, as shown in 4.5a, while the *ready* wave could start before the *resting* wave is completed, as shown in 4.5b.

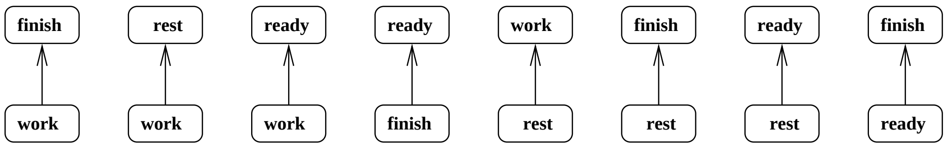


Figure 4.6: Corrective Status Changes. If $P.status$ is incompatible with $P.parent.status$, then $P.status \leftarrow P.parent.status$.

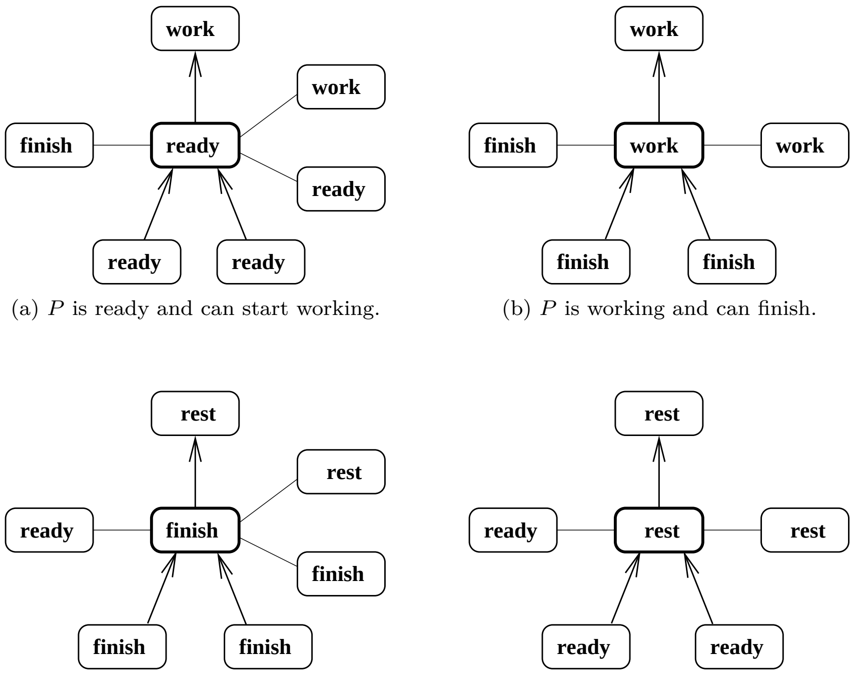
A process P can only execute normally if its status value is not incompatible with its parent's status value. During normal execution of SSBR, the $P.status$ can change only if the status values of the surrounding processes satisfy appropriate conditions, as shown in Figure 4.7.

1. If $P.status = ready$, then $P.status$ is enabled to change to *working* if either P is the root of the BFS tree, or if $P.parent.status = working$, and if in addition, all children of P (in the BFS tree) have status *ready*, and no neighbor of P is resting; as shown in Figure 4.7a.
2. If P is working, then $P.status$ is enabled to change to *finished* if all children of P are finished and all neighbors of P are either working or finished; as shown in Figure 4.7b.
3. If $P.status = finished$, then $P.status$ is enabled to change to *resting* if either P is the root of the BFS tree, or if $P.parent.status = resting$, and if in addition, all children of P (in the BFS tree) have status *finished*, and no neighbor of P is working; as shown in Figure 4.7c.
4. If P is resting, then $P.status$ is enabled to change to *ready* if all children of P are ready and all neighbors of P are either resting or ready; as shown in Figure 4.7d.

Algorithm 4.2 shows the code of SSBR, which is a self-stabilizing emulation of NSSBR. The algorithm takes inputs variables $P.parent$, $P.leader$, and $P.level$, which, if correct, describe a rooted breadth-first search (BFS) tree of the network, where $P.leader$ is the ID of the root process, and $P.level$ is the hop-distance from P to the root (note that the BFS tree is defined using the hop-distance, instead of the weighted distance given as part of the specification of the best reachable problem). SSBR also takes as inputs the function Θ which we are trying to optimize, as well as the order relation "$\succ$" on values of Θ. The sole output variable of SSBR is $P.stable\_best$. Although SSBR runs forever, the value of $P.stable\_best$ is eventually equal to the output of the best reachable problem required by the problem specification.

The local variables of SSBR are $P.status$, $P.best$, $P.dist$, and $P.span$. If the input variable of SSBR are correct, then SSBR will repeat a status wave cycle endlessly. The cycle consists of a broadcast *working* wave, a convergecast *finished* wave, a broadcast *resting* wave, and finally a convergecast *ready* wave. The *ready* wave initializes the local variable of SSBR to match the initial values of the variables of NSSBR, and while a process is working, it emulates the actions of NSSBR. When all processes have completed the emulation of NSSBR, the *finished* wave moves up the tree, followed by the *resting* wave, which then sets $P.stable\_best$ to $P.best$ for all P.

Because of arbitrary initialization, it could happen that $P.stable\_best$ is given the wrong value. But if at least one full status wave cycle has been completed, the value of $P.best$ will be correct at the time



(a) P is ready and can start working. (b) P is working and can finish.

(c) P is finished and can rest. (d) P is resting and can get ready.

Figure 4.7: Normal Status Changes.

the *resting* wave reaches P. Subsequent status wave cycles will not change the value of *P.stable_best*, although the value of *P.best* will change endlessly.

If the input variables fail to specify a BFS tree, then the values of *P.stable_best* could be set to the wrong values many times. However, in that case, one of the processes will detect a *local error* in the BFS tree, and will stop executing actions of SSBR. This "freezing" of that single node will cause SSBR to eventually deadlock. If, at a future time, the input values of SSBR are correct, the deadlock be broken, and SSBR will proceed to compute its output correctly.

4.5.2 Proof of Correctness for SSBR

Lemma 4.8. *Suppose e is a partial execution of* SSBR*, and suppose that during that partial execution, no input value changes. Then, during e, each process P executes a status correction action only finitely many times.*

Proof. By induction on *P.level*. If $P.level = 0$, then either P is not locally correct, in which case P cannot execute at all, or P is a root, in which case its status cannot be incompatible with its parent's status, since it is its own parent, and thus it cannot execute a status correction action.

Suppose $P.level > 0$. If P is not locally correct, then P cannot execute at all. Otherwise, let $Q = P.parent$. By the inductive hypothesis, there will be a configuration γ after which Q will not execute any status correction action. If *P.status* is incompatible with *Q.status*, at γ, then Q is not enabled to change its status, while P is enabled to execute a status correction action, and cannot execute any other action first.

If P executes that status correction action, then no subsequent action by either P or Q can cause *P.status* to become inconsistent with *Q.status*, and hence P will execute no further status correction action. ■

Lemma 4.9. *Suppose e is a partial execution of* SSBR*, and suppose that during that partial execution, no input value changes, and there is one process P that never changes its status. Then, if $Q \in \mathcal{N}_P$,*

Algorithm 4.2 SSBR $(parent, leader, level, \Theta, \succ; stable\_best)$

```
2.1:  for all $P$ do
2.2:    loop {forever}
2.3:      if $P$ is locally correct then {$P$ cannot detect that the BFS tree is incorrect}
2.4:        if $P.status$ is incompatible with $P.parent.status$ then
2.5:          $P.status \leftarrow P.parent.status$

2.6:        else if $P$ is ready then
2.7:          if $P$ is a root or $P.parent$ is working then
2.8:            if all children of $P$ are ready and no neighbor of $P$ is resting then
2.9:              $P.status \leftarrow working$
2.10:           end if
2.11:         end if

2.12:       else if $P$ is working then
2.13:         if $P.span > k$ then {$P.best$ should now be the final value}
2.14:           if all children of $P$ are finished and all neighbors of $P$ are working
                    or finished then
2.15:             $P.status \leftarrow finished$
2.16:           end if
2.17:         else if $P$ can detect that the loop invariant does not hold then
2.18:           $P.span \leftarrow k + 1$ {short-circuit the computation of $P.best$}
2.19:         else if $\forall Q \in \mathcal{N}_P : ((Q.best \succeq P.best) \vee (P.dist + w(P,Q) > k)) \wedge (w(P,Q) + Q.span > P.span)$
               then {iterate the loop of NSSBR }
2.20:           if $\exists Q \in \mathcal{N}_P : Q.best \succ P.best$ and $Q.dist + w(P,Q) = P.span$ then
2.21:             $P.best \leftarrow \max_{\succ} \{Q.best : Q \in \mathcal{N}_P$ and $Q.dist + w(P,Q) = P.span\}$
2.22:             $P.dist \leftarrow P.span$
2.23:           end if
2.24:           $P.span \leftarrow \min \begin{cases} \min \{X.span + w(P,X) : X \in \mathcal{N}_P\} \\ \min \{X.dist + w(P,X) : X \in \mathcal{N}_P \text{ and } X.best \succ P.best\} \end{cases}$
2.25:         end if

2.26:       else if $P$ is finished then
2.27:         if $P.span \leq k$ then
2.28:           $P.span \leftarrow k + 1$
2.29:         else if $P$ is a root or $P.parent$ is resting then
2.30:           if all children of $P$ are finished and no neighbor of $P$ is working then
2.31:             $P.stable\_best \leftarrow P.best$
2.32:             $P.status \leftarrow resting$
2.33:           end if
2.34:         end if

2.35:       else if $P$ is resting then
2.36:         if all children of $P$ are ready and all neighbors of $P$ are resting
                  or ready then
2.37:           $P.best \leftarrow P.\Theta$
2.38:           $P.dist \leftarrow 0$
2.39:           $P.span \leftarrow \min \{w(P,Q) : Q \in \mathcal{N}_P\}$
2.40:           $P.status \leftarrow ready$
2.41:         end if
2.42:       end if
2.43:     end if
2.44:   end loop
2.45: end for
```

Q.status changes at most three times during e.

Proof. Without loss of generality, by Lemma 4.8, no process executes a status correction action during e. Suppose Q.*status* changes infinitely often. Then Q.*status* must follow the cycle $\cdots \to$ *working* $\to$ *finished* $\to$ *resting* $\to$ *ready* $\to$ *working* $\to \cdots$. Whatever the value of P.*status*, there is one value that Q.*status* cannot change to. If P is working, then Q.*status* cannot change to *resting*; if P is finished, then Q.*status* cannot change to *ready*; if P is resting, then Q.*status* cannot change to *working*; and if P is ready, then Q.*status* cannot change to *finished*. Thus, Q cannot change its *status* more than three times, contradiction. ∎

Lemma 4.10. *Suppose e is a partial execution of* SSBR*, and suppose that during that partial execution, no input value changes, and there is one process that does not change its status. Then e is finite.*

Proof. Without loss of generality, by Lemma 4.8, no process executes a status correction action during e. Let P be the process that never changes its *status* during e.
Claim A: Every process P changes *status* only finitely many times during e.
Proof (of Claim A): Let Q be the process that never changes its *status*. We prove the claim by induction on the hop distance to Q. If $P = Q$, we are done. Otherwise, P has a neighbor R which is on the minimum hop-distance path to Q. By the inductive hypothesis, R.*status* changes finitely many times. Let γ be a configuration of e after which R.*status* does not change. By Lemma 4.9, Q.*status* can change at most three times after γ, and hence only finitely many times altogether during e. □

We now continue the proof of Lemma 4.10. By Claim A, after some configuration of e, no value of *status* will change. If P is not working, then P cannot execute any action. If P is working, it can execute at most finitely many actions, since either P.*dist* or P.*span* increases during each action. Thus, e is finite. ∎

Lemma 4.11. *If the BFS tree is correct, then some process is enabled to execute an action of* SSBR*.*

Proof. By Lemma 4.7, every process is locally correct. Assume that P.*status* is not inconsistent with P.*parent*.*status* for any P, since otherwise P is enabled to execute a status correction, and we are done.

Let R be the root of the BFS tree. If R is ready, then all processes are ready, and thus R is enabled to execute Line 2.9 of the code. If R is finished, then all processes are finished, and thus R is enabled to execute Lines 2.31 and 2.32 of the code.

If R is resting, then all processes are finished, resting, or ready. If there exist finished processes, pick a finished node P of minimum level. Then P.*parent* is resting, and all children of P are finished; thus P is enabled to execute Lines 2.31 and 2.32 of the code. If there does not exist a finished process, pick P to be a resting process of maximum level. Then all children of P are ready, and thus P is enabled to execute Lines 2.37 to 2.40 of the code.

If R is working, then all processes are ready, working, or finished. If there exist ready processes, pick a ready node P of minimum level. Then P.*parent* is working, and all children of P are ready; thus P is enabled to execute Line 2.9 of the code. If there does not exist a ready process and there exists a finished process P such that P.*span* $\leq k$, then P is enabled to execute Line 2.28 of the Code. If all processes have *span* $> k$, then pick P to be the working process of maximum level. Then all children of P are finished, and P is enabled to execute Line 2.15 of the code.

The remaining case is that all processes are either working or finished, all finished processes have *span* $> k$, and at least one working process has *span* $\leq k$. By Lemma 4.6, there exists some working process P which satisfies either the condition given in Line 2.17 or the condition given in Line 2.19 of the code, and is thus enabled to change at least one of its values. ∎

Lemma 4.12. *If e is an execution of* SSBR *during which the inputs do not change and the BFS tree is correct, then*

(a) each process changes status infinitely often during e, and

(b) after finitely many steps the values of stable_best stabilize to a solution of the Best Reachable problem.

Proof. By Lemma 4.11, e is infinite. By Lemma 4.9, each process changes *status* infinitely often during e.

Let R be the root process. By Lemma 4.8, we can pick a configuration γ_1 of e after which no process executes a status correction. By Lemma 4.9, there is a configuration γ_2 of e at which R is finished. Since status correction is not enabled, all processes are finished. Similarly, pick a configuration γ_3 of e after γ_2 at which all processes are ready, a configuration γ_4 of e after γ_3 at which all processes are finished. And finally a configuration γ_5 of e after γ_4 at which all processes are ready.

Between γ_2 and γ_3, every process executes Lines 2.37 to 2.39 of the code, and thus, at γ_3, $P.best = P.id$, $P.dist = 0$, and $P.span = \min\{w(P,Q) : Q \in \mathcal{N}_P\}$ for all P. Thus, the loop invariant holds at γ_3.

Between γ_3 and γ_4, SSBR emulates NSSBR, and hence at γ_4, $P.best$ is the best value of $Q.\Theta$ among all Q within distance k of P, for all P, by Theorem 4.1. Then, by the time the execution reaches γ_5, all processes will have executed Line 2.31 of the code, and the output variables of SSBR will be correct. ■

4.5.3 An Example Computation of SSBR

In Figure 4.8, we show an example computation of SSBR for "$\succ$" equals "$<$", "Θ" equals "id" and $k = 30$. In that figure, each oval represents a process P and the numbers on the lines between the ovals represent the weights of the links. To help distinguish IDs from distances, we use letters for IDs. The top letter in the oval representing a process P is $P.id$. Below that, we show $P.dist$, followed by a colon, followed by $P.best$, followed by a colon, followed by $P.span$. In this example we consider that we start from a clean state (Figure 4.8a) and that each node is in a *ready* state (we do not deal with the other states in this example). Below each oval is shown the line of SSBR the process is enabled to execute (none if the process is disabled). An arrow from P to Q indicates that P prevents Q from executing due to conditions Line 2.19.

In Figure 4.8, we show synchronous execution of SSBR. The result would have been the same with an asynchronous execution, but using synchrony makes the example easier to understand.

Consider the process L. Initially it is enabled to execute Lines 2.21, 2.22 and 2.24 (sub-figure (a)). It will, after the first execution (sub-figure (b)), find the value of the smallest ID within a distance of $L.span = 7$, which is D, and will at the same time update its *dist* value to $L.span$, and $L.span$ to $D.span + w(L, D) = 6 + 7 = 13$. As during this step, L has updated its *span* value, $D.span$ is an underestimate of the real *span*, thus D is now enabled to execute Line 2.24 to correct this value. The idea behind the *span* variable, is to prevent the process from searching a minimum ID at a distance greater than *span*. Thus a process will not look at the closest minimum ID in terms of number of hops (as could have done process D at the beginning by choosing process A), but will compute the minimum ID within a radius lower than *span* around itself (hence process D is only able to choose process A in the final step, even if A is closer than B in terms of number of hops).

SSBR halts when $P.span > k$ for all P (sub-figure (i)). In the final step every P knows the process of minimum ID at a distance no greater than k, and $P.dist$ holds the distance to this process.

Sometimes, a process P can be elected clusterhead by another process Q without having elected itself clusterhead (this case do not appear in our example); P could have the smallest ID of any process within k of Q, but not the smallest ID of any node within k of itself. Hence, for solving the k-clustering problem, we need to have a second instance of SSBR that runs with "$\succ$" equal to "$>$" and "Θ equal to "*minid*" to corrects this; it allows the information that a process P was elected a clusterhead to flow back to P.

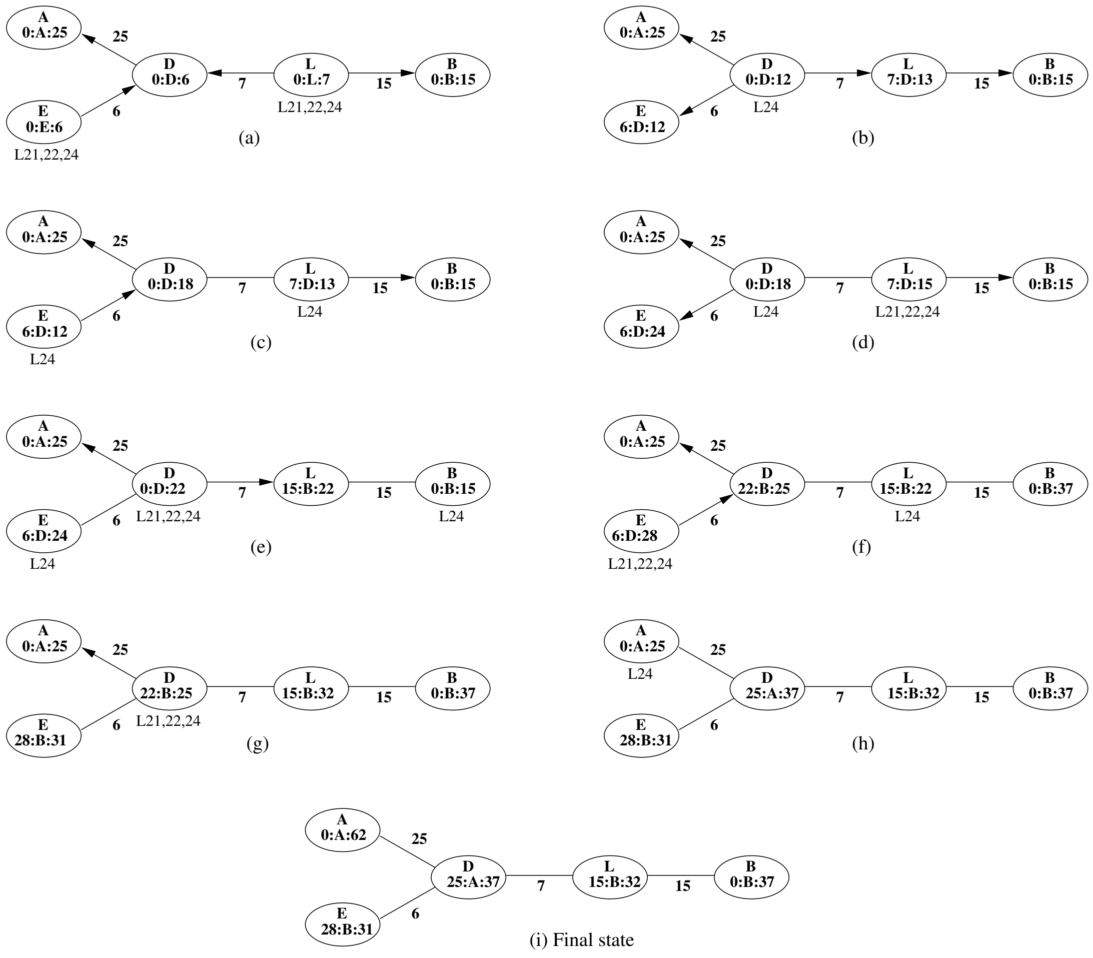


Figure 4.8: Example computation of SSBR for $k = 30$, "$\succ$" equals "$<$" and "Θ" equals "*id*".

4.6 The K-CLUSTERING Algorithm

We now define our combined algorithm, K-CLUSTERING as the combination of four algorithms, as shown in Figure 4.9. Each of these algorithms is self-stabilizing and strongly fair, as defined in Section 4.3. These algorithms are SSLEBFS, two copies of SSBR, and SSCLUSTER, which we define below. The four algorithms are defined as follows:

– A strongly fair and self-stabilizing algorithm, SSLEBFS, which elects a leader and which constructs a BFS tree rooted at that leader. This algorithm outputs variables $P.parent$, the pointer to the parent of P in the BFS tree, $P.leader$, the ID of the elected leader, and $P.level$, the distance from P to the leader. Any algorithm which meets those conditions can be used, such as the one given in [71]: SSLE.

– A copy of SSBR which uses the outputs of SSLEBFS as inputs, and which also uses $P.id$ for Θ and the relation "$<$" for the order relation "$\succ$." The output of this module is the variable $P.minid$, whose correct value is $\min\{Q.id : w(P,Q) \leq k\}$.

– A copy of SSBR which uses the outputs of SSLEBFS as inputs, and which uses $P.minid$ for Θ and the relation "$>$" for the order relation "$\succ$." Thus, the input variables of the third module consist of the output variables of the first two modules. The output of this module is the variable $P.maxminid$, whose correct value $\max\{Q.minid : w(P,Q) \leq k\}$, providing all values of $Q.minid$ are correct.

– The algorithm SSCLUSTER given as Algorithm 4.4 below, which uses *P.maxminid* as its input variable, and has output variables *P.cl_head*, *P.cl_level*, and *P.cl_parent*.

The complexity of K-CLUSTERING comes from the fact that we are dealing with weighted graphs. We need to elect clusterheads that are suitable for the k-clustering problem, *i.e.,* such that any node is at a distance at most k of at least one clusterhead. This is why we need such a complicated arrangement of several algorithms: the usage of SSBR ensures that the elected clusterheads respect this constraint, and the result of SSLEBFS, *i.e.,* the BFS tree, helps for the coordination of the two instances of SSBR. The complexity also comes from the fact that we use only $O(\log n + \log k)$ bits of memory per process.

Algorithm 4.3 is obtained by applying the *Combine* construction, given in Section 4.3, three times; we first combine SSLEBFS with one copy of SSBR, we then combine that algorithm with a second copy of SSBR, and finally combine that result with the algorithm SSCLUSTER. As specified in Section 4.3, our construction needs that, when selected, a process is required to execute an action of each module where that is possible.

Since each of the four modules is strongly fair and self-stabilizing, then K-CLUSTERING is also strongly fair and self-stabilizing, by repeated application of Lemma 4.3. Eventually, the configuration of SSLEBFS will stabilize. After that, the configuration of the first copy of SSBR will stabilize, and after that the configuration of the second copy of SSBR will stabilize. Finally, the configuration of SSCLUSTER will stabilize.

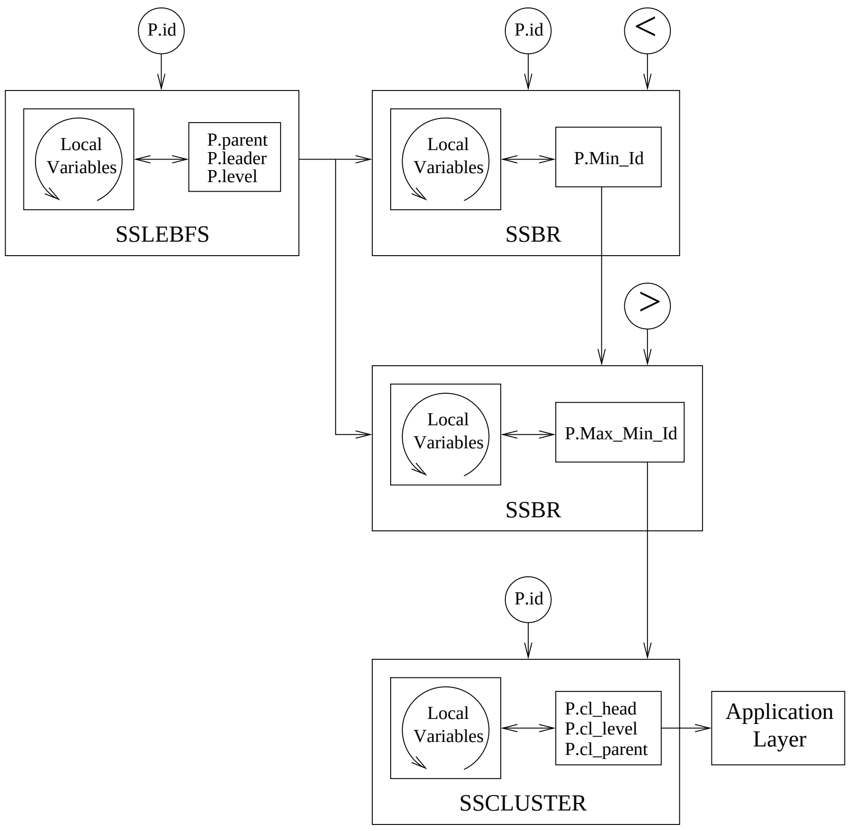


Figure 4.9: K-CLUSTERING is the combination of four strongly fair self-stabilizing algorithms.

Algorithm 4.3 K-CLUSTERING (; *cl_head*, *cl_level*, *cl_parent*)

```
3.1:  for all $P$ do
3.2:    loop {forever}
3.3:      if enabled to do so then
3.4:        Execute an action of SSLEBFS (; $parent, leader, level$)
3.5:      end if

3.6:      if enabled to do so then
3.7:        Execute an action of SSBR ($parent, leader, level, id, <; minid$)
3.8:      end if

3.9:      if enabled to do so then
3.10:       Execute an action of SSBR ($parent, leader, level, minid, >; maxminid$)
3.11:     end if

3.12:     if enabled to do so then
3.13:       Execute an action of SSCLUSTER ($maxminid; cl\_head, cl\_level, cl\_parent$)
3.14:     end if
3.15:   end loop
3.16: end for
```

4.6.1 The Module SSCLUSTER

Algorithm 4.4 updates variables $P.cl\_level$, $P.cl\_head$ and $P.cl\_parent$. If P is a clusterhead, then the variables get respectively values 0, $P.id$ and P. Otherwise $P.cl\_level$ gets the weight of the shortest path from P to the closest clusterhead, $P.cl\_head$ receives the ID of the closest clusterhead (the lowest ID when ties need to be broken), and finally $P.cl\_parent$ gets the neighbor of P that is on the shortest path from P to its clusterhead.

4.6.2 Proof of Correctness for K-CLUSTERING

In order to make use of Lemma 4.3, we must first prove that SSBR and SSCLUSTER have the needed properties.

Lemma 4.13. SSBR *is strongly fair and self-stabilizing.*

Proof. We first prove that SSBR is strongly fair. Suppose that the input of SSBR does not change. We need to prove that either SSBR stops executing, or that every process executes infinitely often.

If the input is correct, then, by Lemma 4.12(a), every process executes an action of SSBR infinitely often.

If the input is incorrect, then, by Lemma 4.7, there is some process P which is not locally correct. Then P will not execute any action of SSBR. By Lemma 4.10, there can be at most finitely many remaining executions of actions of SSBR.

Thus, SSBR is strongly fair.

We now prove that it is self-stabilizing. By Lemma 4.12(b), and the fact that the correct values of $P.stable\_best$ are unique, SSBR is self-stabilizing. Thus, SSBR is strongly fair and self-stabilizing. ■

Given an input configuration of SSCLUSTER, we define a process P to be a *clusterhead* if its ID equals $P.maxminid$, *i.e.,* $P.maxminid = P.id$. We define a *correct* input configuration of SSCLUSTER to be a configuration where every process is within distance k of some clusterhead.

A legitimate configuration of SSCLUSTER is then defined to be a configuration where

1. The input configuration is correct.
2. If P is any process, then $P.cl\_level$ is the distance to the nearest clusterhead.
3. If P is any process, then $P.cl\_head$ is the ID of the nearest clusterhead. In case of a tie, $P.cl\_head$ is the smallest choice.

Algorithm 4.4 SSCLUSTER (*maxminid*; *cl_head*, *cl_level*, *cl_parent*)

```
4.1:  for all $P$ do
4.2:    loop {forever}
4.3:      if $P.maxminid = P.id$ then
4.4:        if ($P.cl\_level \neq 0$ or $P.cl\_head \neq P.id$ or $P.cl\_parent \neq P$) then
4.5:          $P.cl\_level \leftarrow 0$
4.6:          $P.cl\_head \leftarrow P.id$
4.7:          $P.cl\_parent \leftarrow P$
4.8:        end if
4.9:      else
4.10:       if $\exists Q \in \mathcal{N}_P : w(P,Q) + Q.cl\_level \leq k$ then
4.11:         $level \leftarrow \min\{w(P,Q) + Q.cl\_level : Q \in \mathcal{N}_P\}$
4.12:         $head \leftarrow \min\{Q.cl\_head : Q \in \mathcal{N}_P$ and $w(P,Q) + Q.cl\_level = level\}$
4.13:         $parent \leftarrow \min\{Q \in \mathcal{N}_P : w(P,Q) + Q.cl\_level = level$ and $Q.cl\_head = head\}$
4.14:         if ($P.cl\_level \neq level$ or $P.cl\_head \neq head$ or $P.cl\_parent \neq parent$) then
4.15:           $P.cl\_level \leftarrow level$
4.16:           $P.cl\_head \leftarrow head$
4.17:           $P.cl\_parent \leftarrow parent$
4.18:         end if
4.19:       else if $P.cl\_level \neq k + 1$ then
4.20:         $P.cl\_level \leftarrow k + 1$
4.21:       end if
4.22:     end if
4.23:   end loop
4.24: end for
```

4. If P is a clusterhead, then $P.cl\_parent = P$. Otherwise, $P.cl\_parent$ is the neighbor of P of smallest ID that lies on a shortest path from P to $P.cl\_head$.

Note that, for any given correct input configuration, there is exactly one legitimate configuration of SSCLUSTER.

Lemma 4.14. *For any given input configuration, every execution of* SSCLUSTER *is finite.*

Proof. Let e be an execution of SSCLUSTER. During this execution, the values of the input variables of SSCLUSTER do not change, although they may not be correct. Our proof is by contradiction; suppose that e is infinite.

Let $\mathcal{B}$ be the set of process that execute actions of SSCLUSTER only finitely many times during e. Without loss of generality, each member of $\mathcal{B}$ executes no action of SSCLUSTER, since we can start e at the first configuration after all executions of the members of $\mathcal{B}$.

Case I: $\mathcal{B} = \emptyset$.

Let $L = \min\{P.cl\_level\}$, and let $\mathcal{L} = \{P : P.cl\_level = L\}$. When a process $P \in \mathcal{L}$ executes an action of SSCLUSTER, $P.cl\_level$ must increase. Thus, after each member of $\mathcal{L}$ has executed at least once, L must increase. Eventually, $L = k + 1$, which means that no process can execute, contradiction.

Case II: $\mathcal{B} \neq \emptyset$.

For all P, let

$$\Lambda(P) = \begin{cases} P.cl\_level & \text{if } P \in \mathcal{B} \\ \min\{w(P,Q) + \Lambda(Q) : Q \in \mathcal{N}_P\} & \text{otherwise} \end{cases}$$

$$\overline{\Lambda(P)} = \min\{k + 1, \Lambda(P)\}$$

We claim that if $P \notin \mathcal{B}$ and $Q \in \mathcal{N}_P$, and if eventually $Q.cl\_level \leq \ell$, then eventually $P.cl\_level \leq$

$w(P,Q)+\ell$. Let γ be a configuration after which $Q.level \leq \ell$ always holds. The next step where P executes, $P.cl\_level \leq w(P,Q)+\ell$, and $P.cl\_level$ can never decrease below that value.

It follows that $P.cl\_level \leq \Lambda(P)$ for all P, by strong induction on $\Lambda(P)$. If $P \in \mathcal{B}$, the statement holds trivially. Otherwise, there is some $Q \in \mathcal{N}_P$ such that $\Lambda(P) = w(P,Q) + \Lambda(Q)$. By the inductive hypothesis, eventually $Q.level \leq \Lambda(Q)$, and thus eventually $P.cl\_level \leq \Lambda(Q) + w(P,Q) = \Lambda(P)$.

Thus, without loss of generality, $P.cl\_level \leq \Lambda(P)$ for all P and all configurations of e. We now claim that eventually $P.cl\_level \geq \overline{\Lambda(P)}$ for all P. Let $\mathcal{L} = \left\{P : P.cl\_level < \overline{\Lambda(P)}\right\}$, and let $L = \min\{P.cl\_level : P \in \mathcal{L}\}$. If $\mathcal{L} = \emptyset$, the claim holds. Otherwise, $L \leq k$. Every $P \in \mathcal{L}$ is enabled to execute, and when it does execute, $P.cl\_level$ will increase. Thus, eventually, L will increase or $\mathcal{L}$ will become empty. Since L cannot exceed k, $\mathcal{L}$ will eventually be empty, and we have proved the claim.

We can now assume that $P.cl\_level = \overline{\Lambda(P)}$ for all $P \notin \mathcal{B}$ at all configurations of e. Each $P \notin \mathcal{B}$ can then execute at most once, contradicting the infinitude of e. ■

Lemma 4.15. SSCLUSTER *is strongly fair and self-stabilizing.*

Proof. SSCLUSTER is strongly fair by Lemma 4.14. We need only show that it is self-stabilizing.

Assume that the input configuration of SSCLUSTER is correct and never changes. Let $\mathcal{A}$ be the set of clusterheads, namely $\{P : P.id = P.maxminid\}$. Since $P.maxminid$ does not change, $\mathcal{A}$ is fixed.

By Lemma 4.14, we can consider only the last configuration of SSCLUSTER, *i.e.,* a configuration γ where no process is enabled to execute an action of SSCLUSTER. We need only prove that γ is legitimate.

By way of contradiction, assume that γ is not legitimate. Let $Cl\_Level(P)$, $Cl\_Head(P)$ and $Cl\_Parent(P)$ be the correct values of $P.cl\_level$, $P.cl\_head$, and $P.cl\_parent$, *i.e.,* the values those variables must have in a legitimate configuration.

Case I: There is some process P such that $P.cl\_level > Cl\_Level(P)$. Choose that P which has the smallest value of $Cl\_Level(P)$. If $P \in \mathcal{A}$, then P is enabled to execute an action, contradiction. Otherwise, let $Q = Cl\_Parent(P)$. Since $Cl\_Level(Q) < Cl\_Level(P)$, the value of $Q.cl\_level$ is correct, and hence P is enabled to execute, since $w(P,Q) + Q.cl\_level < P.cl\_level$, contradiction.

Case II: Case I does not hold, and there is some process P such that $P.cl\_level < Cl\_Level(P)$. Choose that P which has the smallest value of $P.cl\_level$, and pick $Q \in \mathcal{N}_P$ such that $w(P,Q) + Q.cl\_level$ is minimum. If $w(P,Q) + Q.cl\_level \leq P.cl\_level$, then $Q.cl\_level > Cl\_Level(Q)$ and $Q.cl\_level < P.cl\_level$, contradiction. Otherwise, P is enabled to execute, contradiction.

Case III: $P.cl\_level = Cl\_Level(P)$ for all P, and there is some P for which either $P.cl\_head \neq Cl\_Head(P)$ or $P.cl\_parent \neq Cl\_Parent(P)$. Pick such a P where $P.cl\_level$ is minimum. If P is a clusterhead, then P is enabled to execute, contradiction.
Otherwise, let $\mathcal{Q} = \{Q \in \mathcal{N}_P : P.level = w(P,Q) + Q.cl\_level\}$. By our choice of P, we know that all variables of Q are correct for all $Q \in \mathcal{Q}$. Thus, P will be enabled to execute in order to correct its values, contradiction. ■

Applying Lemma 4.3 twice, we have:

Lemma 4.16. *Eventually* $P.minid = \min\{Q.id : w(P,Q) \leq k\}$ *and* $P.maxminid = \max\{Q.minid : w(P,Q) \leq k\}$ *for all* P.

We then obtain:

Lemma 4.17. *Eventually,* $P.id = P.maxminid$ *if and only if there is some process* Q *such that* $Q.minid = P.id$.

Lemma 4.17 is given in [72]. The proof is by contradiction. If $Q.minid = P.id$ then $w(P,Q) \leq k$ and $P.maxminid \geq Q.minid$. If $P.maxminid > Q.minid$, then for some R, we have $w(P,R) \leq k$ and $R.minid > P.id$, which contradicts the required correctness condition for $R.minid$.

Let $\mathcal{A} = \{P : P.id = P.maxminid\}$, the set of clusterheads. By Lemma 4.17, we know that every process is within distance k of some member of $\mathcal{A}$. By the correctness of SSCLUSTER, and applying Lemma 4.3 once more, we know that K-CLUSTERING partitions the network into cluster, where each process joins the nearest clusterhead. Thus, K-CLUSTERING is correct.

Applying Lemma 4.3 thrice, we have :

Theorem 4.2. K-CLUSTERING *is self-stabilizing, and works under the unfair daemon.*

4.7 Theoretical Bounds

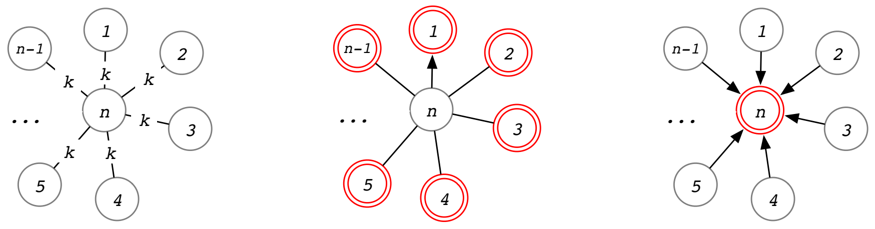
In this section, we give worst case theoretical bounds on the memory consumption of K-CLUSTERING, on its running time complexity, and on the number of clusterheads.

4.7.1 Memory Requirements

K-CLUSTERING uses all the variables of SSLEBFS, in our implementation we used SSLE as presented in [71]. Hence, our algorithm uses all the variables of SSLE, as well as 14 new variables for each process (the value of k, five variables for each copy of SSBR, and three for SSCLUSTER). SSLE needs $O(\log n)$ bits. The variables of ID type can be encoded on $O(\log n)$ bits, distances on $O(\log k)$ bits and the four status on only 2 bits. Hence, on the whole, K-CLUSTERING requires $O(\log n + \log k)$ bits of memory per process. This memory requirement is on the same scale as the optimal memory requirement, as each process has to store at least its ID and the value of k.

4.7.2 Number of Clusterheads

The algorithm can behave very badly compared to the optimal k-clustering, *i.e.,* the clustering with the lowest number of clusterheads. In fact, if OPT_G is the optimal number of clusterheads for a given graph G, K-CLUSTERING can return a solution with $(n-1)OPT_G$ clusterheads. An example of such a bad clustering is given on Figure 4.10: Figure 4.10a presents the initial graph, and Figures 4.10b and 4.10c show the solution returned by our algorithm and the optimal solution, processes with a doubled line are clusterheads, an arrow designates the parent.



(a) Initial graph (b) K-CLUSTERING (c) Optimal result

Figure 4.10: K-CLUSTERING number of clusterheads worst case.

This problem arises because of the distribution of IDs. As our algorithm is comparison based, its solution is constrained by the distribution of the IDs among the processes. Take the same example as the one given on Figure 4.10a, but instead put the lowest ID, 1, on the central node, in this case our algorithm would find the optimal solution.

In order to assess the quality of the solution, one cannot only rely on the $(n-1)OPT$ upper bound, as it is very unlikely that most of the instances of the problem will follow such a bad ID distribution. Moreover, in order to reduce the probability of such a bad configuration, we could add a phase of ID shuffling before running our algorithm. We will see in the simulations presented in Section 4.8, that the number of clusterheads we obtained in the studied cases is lower than the theoretical bound.

4.7.3 Number of Rounds

We now present two theoretical bounds on the number of rounds required by the algorithm to return correct values.

The Chain Graph $G_{n,k}$

In Figure 4.11, we assume that n is even. The network is a chain, *i.e.,* there are edges between P_i and P_j if and only if $|i - j| = 1$. Edge weights are given as follows:

$$||P_i, P_{i+1}|| = \begin{cases} 1 & \text{if } i \text{ is odd} \\ k & \text{if } i \text{ is even} \end{cases}$$

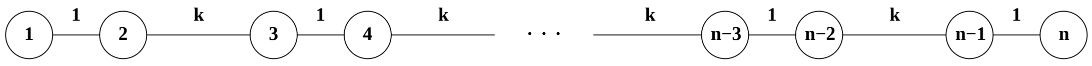


Figure 4.11: The Graph $G_{n,k}$.

Lemma 4.18. *If the algorithm runs on graph $G_{n,k}$, the convergence time is $\Theta(nk)$ rounds in the worst case.*

Proof. For sake of simplicity, we suppose that the processes start in a clean state, *i.e.,* all possible errors have been corrected, and $P.best = P.id$, $P.dist = 0$ and $P.span = \min\{w(P,Q) : Q \in \mathcal{N}_P\}$, note that in this graph for all process P, $P.span = 1$. We only deal with the copy of SSBR in charge of computing $P.minid$.

Initially, only process n is able to execute Lines 2.21, 2.22 and 2.24 due to the condition Line 2.19, and no other process is enabled. Thus, after one round, $n.best = n - 1$, $n.dist = 1$ and $n.span = 2$; and no other process has changed its variables.

During the next $k-1$ steps, only processes n and $n-1$ are enabled to alternatively execute Line 2.24 to update their *span* variable. $n.span$ and $(n-1).span$ are only able to increase by 2 at each step.

Once $(n-1).span > k$, $n-1$ is enabled to execute Lines 2.21 and 2.22, and set $(n-1).best = n-2$ and $(n-1).dist = k$. Then, process $n-2$ is enabled to execute Lines 2.21, 2.22 and 2.24, which starts a new cycle of $k-1$ rounds between $n-2$ and $n-3$ to update *span*.

These update cycles are repeated until a cycle reaches process 2, which is the last cycle between 1 and 2: the processes can only be updated following a descending order on their IDs.

Overall, it requires $(1+(k-1))\times n/2 = kn/2$ rounds to complete the execution of SSBR for computing $P.minid$. Hence the $\Theta(nk)$ bound. ■

The Random Graph $R_{n,k}$

Assuming n is even, we construct the graph $R_{n,k}$ as follows. The nodes of $R_{n,k}$ are the integers $\{1, \ldots n\}$. We randomly partition the processes into pairs, which we call *special pairs*, in such a manner that all such partitions are equally likely. If $\{i, j\}$ is a special pair, we write $partner(i) = j$ and $partner(j) = i$. We say that i is *superior* if $partner(i) < i$; otherwise we say that i is *inferior*. $R_{n,k}$ is complete, and for any nodes i and $j \neq i$, the weight of the edge between i and j is 1 if $j = partner(i)$, and k otherwise. Figure 4.12 presents an example of such a graph.

Lemma 4.19. *If the algorithm runs on graph $R_{n,k}$, the convergence time is $O(nk)$ rounds in the worst case.*

Proof. For sake of simplicity, we suppose that the processes start in a clean state, *i.e.,* all possible errors have been corrected, and $P.best = P.id$, $P.dist = 0$ and $P.span = \min\{w(P,Q) : Q \in \mathcal{N}_P\} = 1$. It takes $n/2$ steps for processes to have the following values: $P.best = P.id$ if P is inferior, or $partner(P)$ if P is superior; $P.dist = 0$ if P is inferior, or 1 if P is superior; $P.span = 1$ if P is inferior, or 2 if P is superior.

In fact it does not take exactly $n/2$ steps, but $n/2$ executions of Lines 2.21, 2.22 and 2.24, but even if these steps do not occur consecutively, on the whole it will take $n/2$ steps for all processes to pass

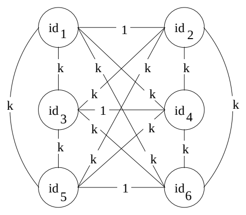


Figure 4.12: The Graph $R_{n,k}$.

through this configuration. Then, only two processes will be able to update their *span* variable, due to the $P.dist+w(P,Q) > k$ condition, which is not true for most of the processes (for those $P.dist+w(P,Q) = k$), and $Q.best \geq P.best$. Hence, only the special pair with the highest *id* can update, and only one process is enabled at each step. This is verified until this special pair finds 1 as the best *id*, which takes k steps. We repeat this for the next special pair having the second highest *id*, and so on and so forth... Hence, as we have $n/2$ special pairs, on the whole it takes $O(nk)$ rounds. ■

4.8 Simulations

We designed a simulator to evaluate the performance of our algorithm [4]. In order to verify the results, a sequential version of the algorithm was run, and all simulation results compared to the sequential version results. Thus, we made sure that the returned clustered graph was the correct one. In order to detect when the algorithm becomes stable and has computed the correct clustering, we compared, at each step, the current graph with the previous one; the result was then output only if there was a difference. The stable result is the last graph output once the algorithm has reached an upper bound on the number of rounds (we set this number at least two orders of magnitude higher than the convergence time of the algorithm). The unfairness is simulated as follows. The daemon chooses randomly an enabled process and prevents it from executing any action, until it becomes the only process that can execute.

4.8.1 Effect of the k Value

Example of the graph presented in Figure 4.13. We ran the simulator on the weighted graph illustrated in Figure 4.13. For each value of k, we ran 10 simulations starting from an arbitrary initial state where the value of each variable of each process was randomly chosen. Hence the processes do not start in a clean state.

Figure 4.14a presents the number of clusterheads found for each run and each value of k. As K-CLUSTERING returns exactly the same set of clusterheads whatever the initial condition, the results for a given k are all the same. Note that the number of clusterheads decreases as k increases, and even if the algorithm may not find the optimal solution, it gives a clustering far better than a naive $O(1)$ self-stabilizing algorithm which would consists in electing each process a clusterhead. The figure shows that the number of clusterheads quickly decreases as k increases.

Figure 4.14b presents the number of rounds required to converge. This figure shows two kinds of runs: with an unfair daemon that holds a random process until no other process is able to execute, and with a fair daemon that selects every enabled process at every step. As can be seen, the number of rounds is far lower than the theoretical bound $O(nk)$, even with an unfair daemon.

Random graphs. We also generated 50 random graphs containing each 100 nodes, with edges weight uniformly taken between 1 and 100. Figure 4.15a, and 4.15b present respectively the number of clusterheads, and the number of rounds. As can be seen the number of rounds (represented in log-scale) is several order of magnitude lower than the theoretical bound, and the number of clusterheads quickly decreases. Each type of points on the graphs represent a given platform.

4. It can be found at `http://graal.ens-lyon.fr/~bdepardo/down_files/k-clustering/k-clustering.bz2`, the file also contains all the platforms and the results.

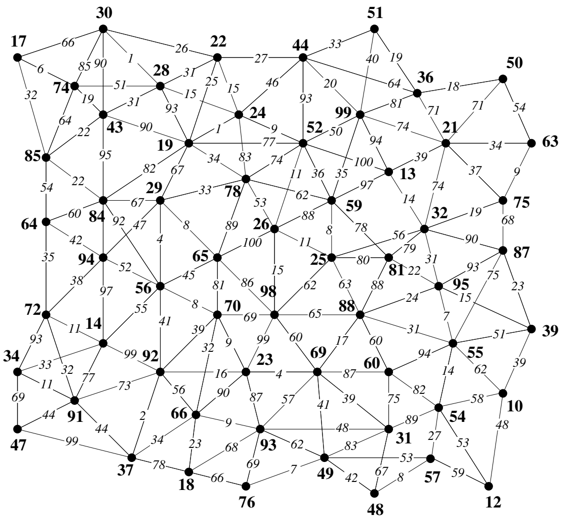
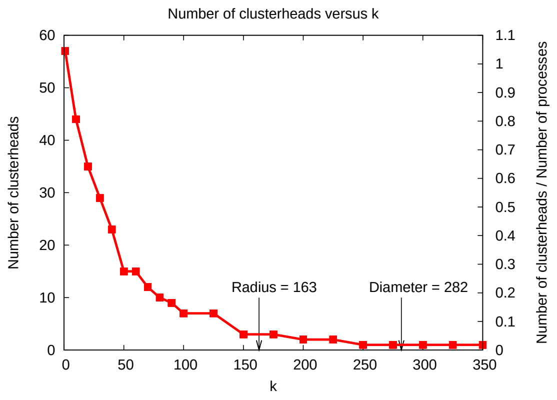
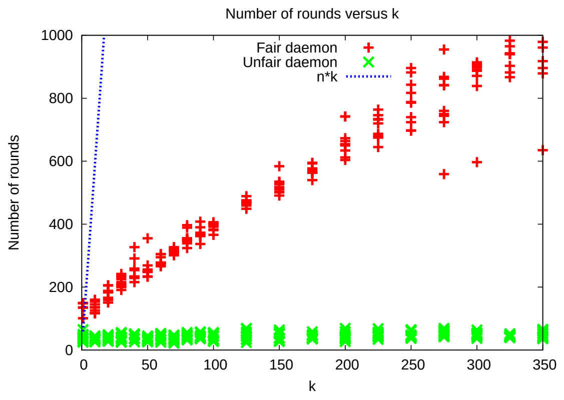


Figure 4.13: Example graph: *number of nodes* = 59, *diameter* = 282, *radius* = 163, weights between 1 and 100.



(a) Clusterheads



(b) Rounds

Figure 4.14: Number of clusterheads, and number of rounds for graph Figure 4.13.

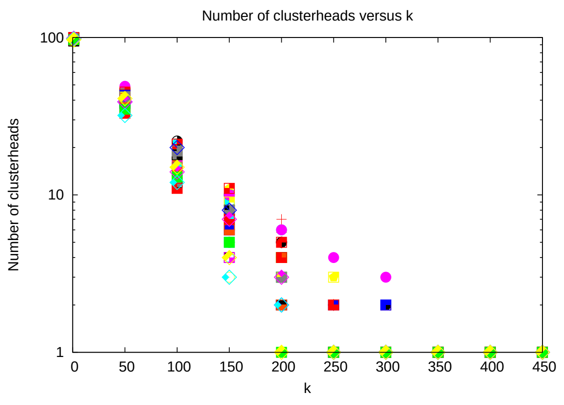
4.8.2 Complexity Bounds

Graph $G_{n,k}$. The number of clusterheads obtained for each instance of the graph is $n-1$: every node is elected clusterhead, apart from node n which connects itself to node $n-1$.

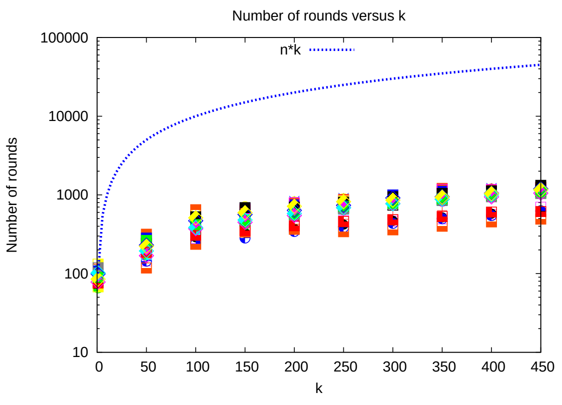
Figure 4.16a presents the number of rounds obtained with and without unfairness for different values of n, for $k = 100$. It can be observed that the number of rounds follows the theoretical bound $O(nk)$.

Graph $R_{n,k}$. The number of clusterheads obtained for each instance of the graph is 1: every node connects itself to the node of lowest ID: 1.

Figure 4.16b presents the number of rounds obtained with and without unfairness for different values of n, for $k = 100$. It can be observed that the number of rounds follows the theoretical bound $O(nk)$.

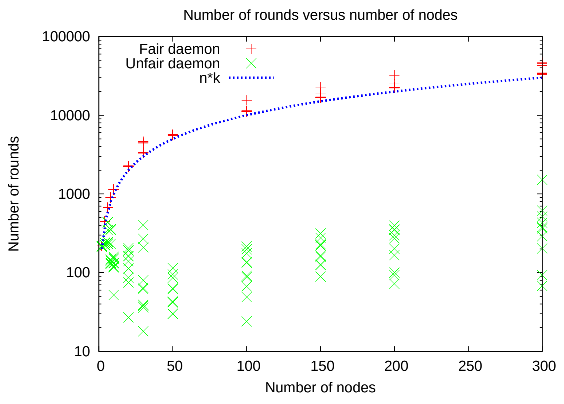


(a) Clusterheads

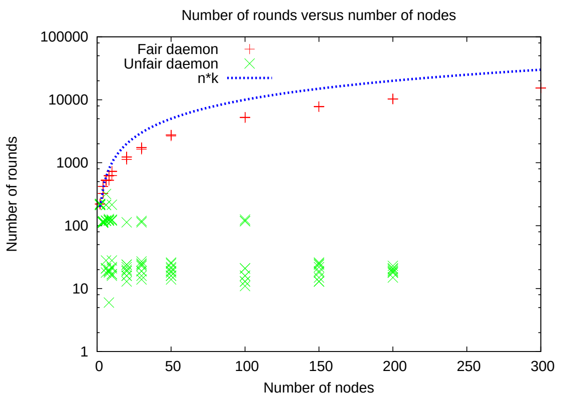


(b) Rounds

Figure 4.15: Random graphs: number of clusterheads and number of rounds.



(a) $G_{n,k}$



(b) $R_{n,k}$

Figure 4.16: Number of rounds for graphs $G_{n,k}$ and $R_{n,k}$.

4.9 Conclusion

In this chapter, we have presented a self-stabilizing asynchronous distributed algorithm for construction of a k-dominating set, and hence a k-clustering, for a given k, for any weighted network. In contrast with previous works which dealt with unweighted graphs, or weights on the nodes, our algorithm deals with an arbitrary metric on the network, *i.e.,* weights on the links, and hence, is able to take into account more realistic communications cost. Note however that our approach could easily be extended to take into account weights on nodes, as we could consider the nodes weight instead of their IDs to elect the clusterheads. To the best of our knowledge, this work is the first solution to the k-clustering problem on weighted graphs.

We first gave conditions under which the combination of self-stabilizing algorithms is also self-stabilizing under the unfair daemon. Then, we presented a non self-stabilizing generic distributed algorithm for solving the *Best Reachable* problem. This latter is then extended so as to become self-stabilizing. For this purpose, our solution uses the principle of *broadcast and convergecast waves.* Finally, we present our self-stabilizing algorithm, K-CLUSTERING, for solving the k-clustering problem. K-CLUSTERING, is the combination of four strongly fair self-stabilizing algorithms: SSLEBFS, SSBR (used twice) and SSCLUSTER. It executes in $O(nk)$ rounds, and requires only $O(\log n + \log k)$ space per process. A comprehensive correctness proof of the algorithm is provided, but also simulation results highlighting the fact that K-CLUSTERING executes in a reasonable number of rounds, despite its bad theoretical complexity.

Part III

Mapping Tasks

Chapter 5

Preliminaries on Scheduling

Once a mean of accessing the computing power of a grid has been deployed, still remains the problem of scheduling the jobs onto the resources. On such a platform, several users compete for the execution of their tasks and thus we need scheduling strategies to deal with the work to be done. In this chapter, we give an overview on scheduling models and techniques. We present in Section 5.1 the different application models. Then, we give in Section 5.2 some of the optimization objectives commonly used in scheduling. Finally, we present approaches commonly used for designing scheduling algorithms in Section 5.3.

5.1 Application Models

Many kinds of applications exist and are executed on computing grids. Some are monolithic, some need to be executed on several processors either because they use a parallel programming paradigm (such as MPI), or because they are composed of several sub-tasks interconnected by precedence constraints. In this section we do not deal with applications based on parallel programming paradigm.

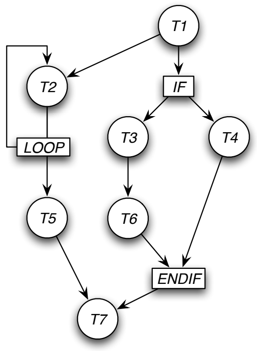
5.1.1 Directed Acyclic Graph

The most general model for representing an application is the *workflow*. A workflow is a directed graph representing the dependencies between tasks of an application. Apart from direct dependencies, it can also contain elaborated control structures such as conditional instructions as well as loops. Depending on the data values, and the control structures, the number of instantiated tasks can vary, and may only be determined during execution of the workflow. Figure 5.1a depicts a workflow with a loop and a conditional branch.

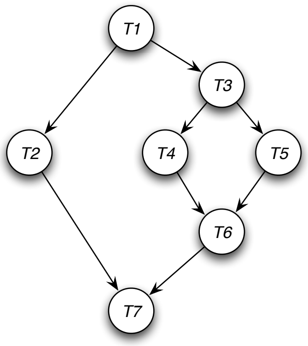
However, most of the applications can be depicted by a simpler workflow, which does not contain any conditional instructions, a *dataflow*. A dataflow can be represented by a *Directed Acyclic Graph* (DAG). A DAG explicitly contains all the tasks that need to be executed, as well as their dependencies. Thus, the number of tasks is known prior to execution. Dependencies correspond to data transfer between tasks. Figure 5.1b depicts a DAG with seven tasks, and finally, Figure 5.1c shows the DAG for a complete cosmological simulation.

5.1.2 Embarrassingly Parallel Applications

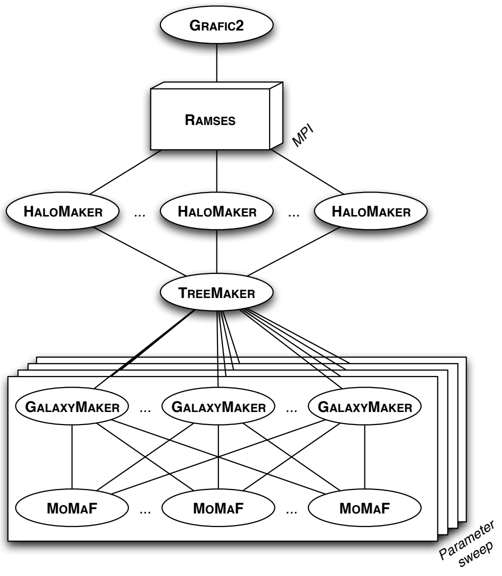
In the next chapter, we focus on a less general kind of applications, but nevertheless frequently used: applications whose sub-tasks can be executed in any order, and even concurrently. Applications which do not require any control structures, nor have dependencies between tasks, are called *trivially parallel applications*. Two models are generally used to depict such applications.



(a) Workflow



(b) DAG



(c) Cosmological simulations DAG

Figure 5.1: Workflows, and Directed Acyclic Graphs.

Divisible load

The *divisible load* model reflects applications that can be divided into an arbitrary number of parts. We can consider that the processing time of an application can be divided into *chunks* of arbitrary size, containing a set of unitary computations. These latter can be considered small compared to the total computation requirements to process the whole input. Each chunk can then be processed independently on different processors, and in any order. This model represents applications that can be perfectly parallelized.

In practice, this model is a reasonable relaxation for applications made up of fine grain tasks without any dependencies between them. Several scientific applications can fit into this models. Their execution time often depends on the size of the input data. Thus, dividing the load into small chunks often consists in dividing the input data. For example comparison of DNA sequences in biology [29], or image processing [123] fit into this model.

The divisible load has been popularized by the book written by Bharadwaj, Ghose, Mani, and Robertazzi [48], and widely studied since then, see for example [41, 144]. The success of the divisible load theory comes from the fact that its formulation allows to leverage the problems of having $\mathcal{NP}$-complete problems. It provides a background to easily define theoretically optimal scheduling algorithms.

Bags-of-tasks

Another well studied model of application, is the so called *bag-of-tasks* model. It is somewhat similar to the divisible load model, as we consider that each application is composed of a given number of independent sub-tasks. Each of these sub-tasks is supposed to be computed atomically. In this model, the size of the sub-tasks is fixed, contrarily to the divisible load model where we can choose arbitrarily their sizes, and the sub-tasks of a given application share the same characteristics.

In practice, many applications follow this model. Indeed, parallel applications requiring several processors to be executed are more prone to errors, as the crash of one processor often means the crash of the whole application. Thus, many applications are designed to run on a single processor. This class of applications is typical of the grid infrastructures' usage. Parameter sweep applications [62] fall into this category, such as the ones studied in the following part of this dissertation, in Part IV: cosmological simulations' post-processing applications [50, 106], *e.g.,* the whole DAG depicted on Figure 5.1c is not necessarily executed all at once, and applications such as GalaxyMaker and MoMaF are often executed several times with various parameters, in this case, the bag-of-tasks model is well suited. Other applications such as the ones running on desktop grids like in the BOINC project [33], and SETI@home [34] also fit in this model.

The bag-of-tasks model is the one used in the next chapter, though we will also refer to the divisible load model.

5.2 Objectives

Many different schedules can be obtained from the same set of tasks. The differences in the results may of course come from the used algorithms, but are particularly influenced by the *objective* the algorithms tend to optimize. Several "base" metrics exist, and most often, optimizing one is at the expense of some others.

5.2.1 Based on Completion Time

The first metric that comes to mind when we want to schedule a task, is its completion time. When several tasks compete for resources, we talk about *makespan* of a schedule to refer to the maximum of the tasks' termination time. This is the most commonly used metric in the literature. Optimizing the makespan amounts to optimizing the platform utilization, and is thus system-centric. Indeed, taking into account only the maximum of the tasks' completion time, does not give any information on the completion time of each task taken individually. Thus, this approach can be used when all tasks have the same importance, and when they belong to the same user. When several users compete for the resources, the *average completion time* can be used instead.

When a great number of independent tasks are submitted to the system, the makespan metric can be relaxed, and instead the *throughput* can be considered, *i.e.,* the number of tasks that are executed per time-unit. This metric served as a base metric in Part I to qualify the performance of a Diet hierarchy.

5.2.2 Based on Time Spent in the System

When tasks are not released all at the same time, but instead arrive over time, the metrics based on completion time are no longer relevant. Indeed, what would be the point in optimizing the makespan if among all the tasks to be scheduled, one is released so late, that all the other tasks have enough time to be computed. In this case any schedule would be optimal.

Instead, metrics based on the time spent by the tasks in the system, *i.e.,* the *flow time*, may be used, such as *sum-flow* or *max-flow*. The flow time is the difference between the completion time of a task and its release date, *i.e.,* the execution time of the job, plus its waiting time.

The problem with flow time based metrics is that they tend to favor longer jobs, and can even lead to starvation. Thus arose new metrics based on the *slowdown* a job experience when scheduled on a loaded system, compared to when scheduled on an unloaded system. This metric is called the *stretch* [44], also referred to as the *slowdown* [88]. The stretch is defined as the ratio between the flow time, and its required processing time in a homogeneous platform context. Thus in our context of heterogeneous platforms, the slowdown is the ratio between the flow time, and the runtime experienced on the system if the task would have been the only scheduled task.

5.2.3 Based on Energy Consumption

As computing infrastructures tend to become larger and larger from day to day, the total energy consumed to by these infrastructures becomes higher and higher. Thus, scientists recently focused on another metric which is the power consumption of a schedule [156]. Two main approaches exist to minimize the consumed power: *dynamic power scaling*, and *power management*. The first one relies on dynamically scaling the processor's voltage [171]. The power consumption increases with the processor's voltage, but conversely the performance of the processor decreases when the voltage is reduced. The second one consists in shutting down a processor (or any other hardware part) when idle [66].

5.3 Scheduling

5.3.1 Scheduling Models

There exists several ways of tackling a scheduling problem. Three main scheduling models can be found in the literature, the difference between them being the knowledge the scheduler has about the applications. In the simplest model, we know *beforehand* all the characteristics of the applications (computation size, number of tasks, *etc.*), and all tasks are available and ready to be scheduled. In the *offline* model, everything about the applications is also known beforehand, but each task can have its own release date, which is also known from the start. If applications' characteristics are not known by the scheduler before their release dates, and the release dates are also unknown, then the model is said to be *online*. We can also have in-between models, such as the semi-clairvoyant model, when only partial information is known about the future.

5.3.2 Difficulty

Once the scheduling problem has been clearly identified (*i.e.,* models and objective have been chosen), we need to provide a way of solving it. Sadly, scheduling problems tend to be difficult to solve. Many of them are $\mathcal{NP}$-completeness. Even the simple problem of scheduling independent tasks on two identical processors, while minimizing the makespan, is already $\mathcal{NP}$-complete [65]: the proof is easily done with a reduction from 2-partition [97]. We can either propose a formulation of the problem that can lead to an optimal solution, but at the expense of maybe spending lots of time computing the solution, or rely on heuristics to provide an approximation of the optimal result, or a "hopefully good" solution.

5.3.3 Approaches

There exist several approaches to deal with a scheduling problem. Apart from designing heuristics, a commonly used approach that helps leveraging the $\mathcal{NP}$-completeness problem is to rely on *Steady state scheduling*. Steady state scheduling has been introduced by Bertsimas and Gamarnik in [47]. It is a technique to leverage the complexity of scheduling tasks with the objective of minimizing the execution time. This method is applicable whenever we have a large number of tasks to schedule, and consists in neglecting the cost of initialization and clean-up phases in the scheduling, to concentrate only on the period in which the system works to it "maximum capacity". The integer formulation of the problem can be relaxed to rational formulation, *i.e.,* instead of considering the exact scheduling of the tasks, we aim at obtaining a partial mapping of the tasks on the processors that leads to an optimal schedule. The precise scheduling of communications and computations arises from the quantities of computations and communications obtained with the linear program. It consists in finding a period on which all quantities can be turned into integers. This approach is the one we used in Part I of this dissertation: we did not care about the precise requests scheduling, nor about the starting and ending phases, but instead we studied the behavior when the platform worked at its fullest.

The other commonly used approach relies on divisible load theory, which often removes the $\mathcal{NP}$-completeness of the problem. However, in the problem presented in the next chapter, we do not consider having a sufficiently large number of tasks to neglect the initialization and termination phases, nor do we consider that we have applications that can fit in the divisible load model. Thus, we rely on a method consisting in describing our scheduling problem with a linear program (this approach has already been introduced in Section 1.3.2). This approach consists in describing with sets of linear equations or inequations, all the constraints the scheduling problem has to fulfill in order to be valid.

In Chapter 6, we tackle a scheduling problem relying on the beforehand and offline models. We propose a formulation of the problem relying on a linear program formulation that leads to an optimal solution.

Chapter 6

Scheduling with Utilization Constraints

Once a mechanism to access the machines is provided to the users, remains the problem of efficiently selecting the resources on which the jobs are executed. Studying scheduling problems corresponds to having an even closer look at the computation process: we began by studying at a high level grid computing with middleware deployment and machines clustering, now we deal with the effective tasks' execution.

Scheduling problems and deployment problems are in fact intimately related. Whereas obviously deployment has an impact on scheduling results (the more machines we have, the more scheduling possibilities we have), scheduling results can also have an impact on deployment. Indeed, analyzing scheduling results might lead to detect that too few (or too many) machines are managed by the middleware, and thus that modifications in the middleware deployment might be necessary. A control loop between deployment and scheduling might thus appear, which leads to dynamic adaptation of the resources' access mechanisms.

In this chapter, we only concentrate on a scheduling problem, the dynamic adaptation of a middleware being part of our future work. We deal with the problem of scheduling several bags-of-tasks on a platform where access to resources can be limited. Two related problems are addressed in this chapter:

1. **Without release dates.** We first deal with the case where tasks are released all at the same time, and we aim at scheduling them on a given period of time, during which the computing power utilization might be limited. The objective of this schedule is to try to map as many task as possible from each bag-of-tasks, while respecting a certain "fairness."
2. **With release dates.** Then, we tackle the same problem, but when tasks can arrive at anytime, *i.e.,* release dates are considered. In this case, we do not consider having only one period of time on which the jobs must be scheduled, but instead we try to map them such as the maximum stretch is reduced. So many periods, on which the utilization of the resources are limited, are considered.

We give theoretical results, based on linear program formulations, to solve these problems. We iteratively build the solutions. Starting from simple linear program formulations, we point out the encountered problems, and present how they can be overcome.

The rest of this chapter is organized as follows. In Section 6.1, we present the problem addressed in this chapter. Then, we tackle two different versions of the problem, with different objectives. The first version supposes that all tasks are released at the same time, and the goal is to schedule as many tasks as possible. This problem is tackled in Section 6.2. The second one assumes that tasks can arrive at any time. The problem is thus seen from a different angle, and the goal is no longer to maximize the number of scheduled tasks, but to minimize the stretch they are subjected to. This problem is presented in Section 6.3.

6.1 Problem Statement

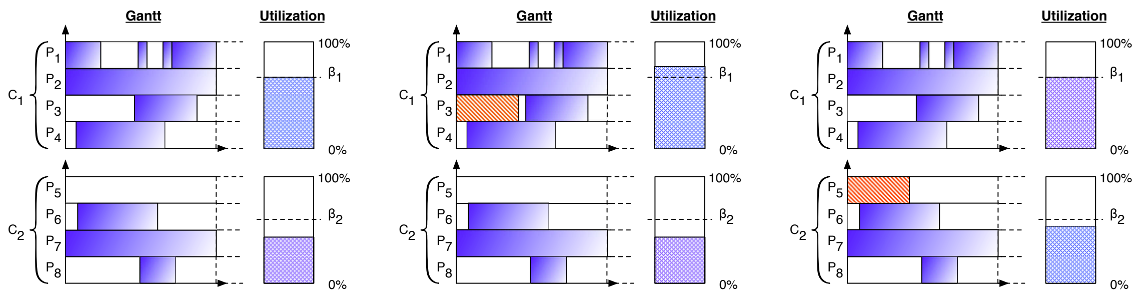
We are dealing with the problem of scheduling collections of independent and identical tasks on a heterogeneous platform which already contains scheduled tasks. In this chapter, we won't take into account the communication costs that would occur for transferring the input data on the platform, retrieving the output data, or even to send the application on the platform. We will consider that required software are already in place on the platform, and that input and output data are either small enough to neglect their transfer time, or that input are already present and that output data stay in place.

6.1.1 Platform

We consider a platform which is an aggregation of clusters, as is often the case in grid computing. Each cluster is composed of a set of resources, where several users compete to have access to resources. This access can be granted in two ways. Either a job is submitted along with its description, and its execution time, then it is scheduled by the batch scheduler. Or, its access can be granted through reservations, *i.e.,* a user submits a job, along with its description, and states that this job should start running at a given date, and for a given period of time. Thus, whenever a task is submitted, its effective scheduled is conditioned by the already existing tasks: for each cluster, and each processor, we can have a Gantt of already scheduled tasks.

What if, for each available cluster, a *maximum cluster utilization percentage* is given to the users? Stated differently, what if the users are allowed to schedule jobs on a cluster if and only if this cluster has not yet reach a fixed utilization percentage on a given period of time? This limitation ensures that the machines are not fully utilized on a given time period. What is the point in having underused machines? Several reasons may lead to this limitation. This percentage could be set by administrators for maintenance and administrative reasons. It could also be imposed by a contract to guaranty a certain quality of service for a selected group of users (those who aren't subjected to a tight resource utilization limitation), or if the pricing varies with the resources utilization, then it might be a will from the user not to use a cluster when its loaded, so as not to pay too much. It could also be used for energy saving, to enforce the use of clusters that are for example on cold parts of the globe, or set periods of limitations such that the computations are mainly executed during the night,...

More formally, our platform is composed of m processors, $P_i, 1 \leq i \leq m$, which are grouped into p clusters, $C_j, 1 \leq j \leq p$. We will denote by $\gamma_i^j \in \{0, 1\}$ the fact that a processor P_i belongs, or not, to the cluster C_j ($\gamma_i^j = 1$ if P_i is in C_j), or to simplify the notation, we will denote by $\{P_i : P_i \in C_j\}$ the set of processors of C_j. Each cluster C_j has, on a given period of time T, a maximum utilization percentage β_j. This β_j constraint has to be respected: whenever a new task is scheduled on the cluster, the new load of the cluster cannot work more than β_j of its maximum workload capacity, see Figure 6.1.



(a) Initial Gantt

(b) Trying to add one task on C_1: β_1 is exceeded

(c) Trying to add one task on C_2: β_2 is not exceeded

Figure 6.1: Maximum utilization percentage example.

Each processor P_i has a given computation power w_i, *i.e.,* the work that can be done per time unit (expressed in Mflops). However, as on the given period T other tasks might be scheduled, P_i already has an utilization percentage α_i, *i.e.,* on the period T, P_i can offer at most $w'_i = (1 - \alpha_i)w_i$ units of work. This utilization percentage reflects the workload incurred by tasks already scheduled on the processor: we already have a Gantt of scheduled tasks and available *slacks* (periods where the processor does not have any task scheduled).

6.1.2 Applications

We envision a situation where users, or clients, submit several *bags-of-tasks* applications to a heterogeneous platform. This is a common utilization of grids, where users wish to study parameter sweep applications for example. Users aim at maximizing the number of tasks scheduled for each bag-of-tasks in the T period.

We consider that we have q bags-of tasks $a_k, 1 \leq k \leq q$. A bag-of-task a_k contains f_k tasks of the same application (for the sake of simplicity a_k will refer either to the bag of task or the application itself), each of these tasks requires d_k units of work to be computed (expressed in flop). We also consider that all applications require only one node to be executed, thus we do not deal with parallel applications. We also denote by an integer variable $\delta_i^k \in [0, f_k]$ the number of instances of a_k scheduled on processor P_i.

As stated above, we consider that the applications do not require any data to be transferred. This amounts to have the application already present on certain processors. Let E_i^k be a Boolean variable, stating whether or not an application can be executed on a processor: if $E_i^k = 1$ then application a_k can be executed on processor P_i, $E_i^k = 0$ otherwise. This of course limits the scheduling possibilities, as we will only be able to schedule a task of a_k on a processor i such that $E_i^k = 1$.

6.1.3 Why single node jobs?

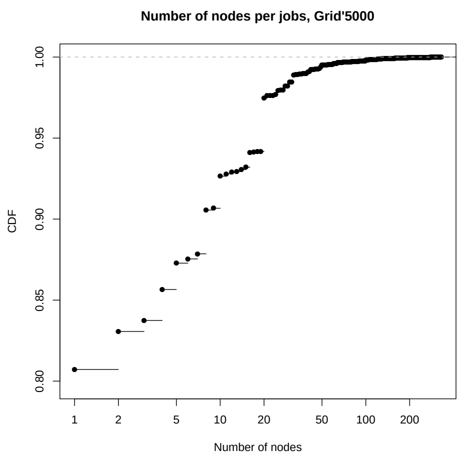
Why do we concentrate on tasks requiring only a single processor to be executed, while there exist many parallel applications? It turns out that grid workloads are dominated by single-node jobs [109]. Figure 6.2 presents the Cumulative Distribution Function (CDF) of the number of processors per job for different platforms: DAS-2 [3], Grid'5000 [51], NorduGrid [86] and SHARCNET [22]. Data has been extracted from OAR [132] databases for Grid'5000, and "The Grid Workload Archive" [9] for the DAS-2, NorduGrid and SHARCNET platforms. As can be seen, on Grid'5000 80% of the jobs used only one node during the period 2008-2009, on SHARCNET more than 90% during the period 2006-2007, on NorduGrid 99% of the jobs during the period 2003-2006. Only on DAS2 less than 40% of the jobs use only one node. This shows that a majority of the jobs submitted to grids are not parallel jobs. One restriction though on this analysis, is that we do not have the information if these jobs are really stand-alone jobs, or if they are part of workflows.

6.1.4 Goals

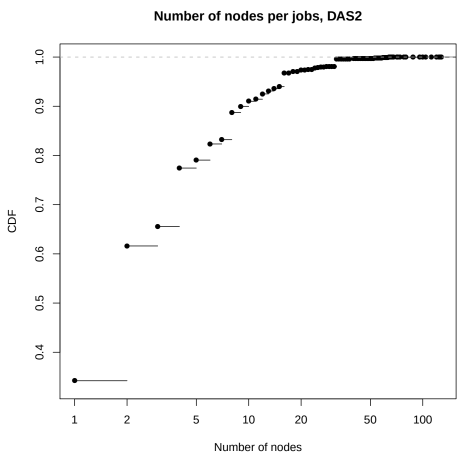
As presented in Chapter 5, several objectives can be used to schedule tasks. As we have several bags-of-tasks to schedule, which may belong to several users, we aim at providing them a notion of *fairness*. As we deal with two versions of the problem, we define two notions of fairness as follows.

In Section 6.2, we will consider the case of scheduling tasks without release dates on a fixed period of time T. As on this period, it may not be possible to schedule all the tasks, our goal is then to schedule as many tasks as possible within each bag-of-tasks. However, taking care only of the total number of scheduled tasks may lead to bias. Indeed, tasks requiring few computations would be privileged, as they would be easier to schedule. Moreover, as each bag-of-tasks may belong to several users, our "fairness" metric will be to offer almost the same ratio of scheduled tasks to requested tasks per bag-of-tasks. Thus, for each $a_k, 1 \leq k \leq q$, we will aim at maximizing $\frac{\sum_{i=1}^{k} \delta_i^k}{f_k}$.

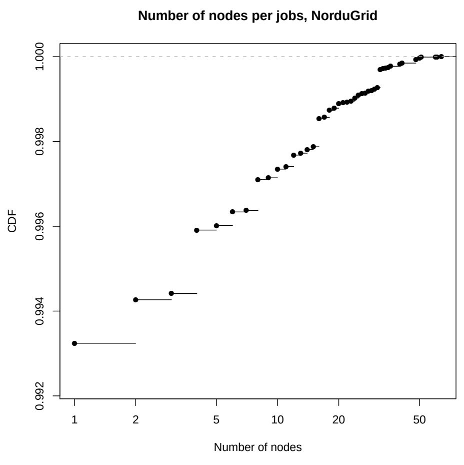
On the other hand, in Section 6.3, we consider the problem of scheduling tasks with release dates, but not on a fixed period T: there would be no point in trying to schedule tasks that may have release dates outside the given period? Trying to optimize the above mentioned ratio would be pointless. Hence, the "fairness" metric we consider is the maximum stretch the tasks will receive. This metric ensures the users, that if the maximum stretch is $\mathcal{S}$, then their tasks won't stay more than $\mathcal{S}$ times the best time they would have spent in the system if alone on the platform. Thus, we will aim at minimizing this maximum stretch, so as to provide the users an upper bound on the degradation on the performance of the tasks' execution time.



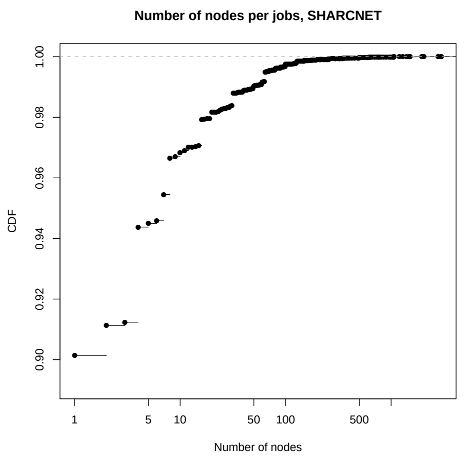
(a) Grid'5000, 2008-2009



(b) DAS-2, 2005 - 2006



(c) NorduGrid, 2003 - 2006



(d) SHARCNET, 2006 - 2007

Figure 6.2: CDF of jobs' number of nodes for various grids.

6.2 Scheduling Jobs without Release Dates

We first consider that all tasks are released all at once, at the beginning of the T period. We aim at scheduling the maximum number of instances of each type of application within this T period.

Our problem cannot be dealt with a steady-state approach, this for two reasons. First of all, we do not have dedicated resources. We already have tasks scheduled on the processors, this prevents the use of a steady-state approach which gives a cyclic schedule. Secondly, we do not suppose that the number of tasks for each application is big enough to reach a steady-state.

We iteratively build a solution to our problem based on a linear programming formulation. We first present a linear program for scheduling the tasks on the processors, while respecting the maximum cluster utilization constraints, but without taking into account real slacks on the processors. Secondly, we add new constraints to obtain a valid schedule.

6.2.1 A Linear Program for "Rough" Allocation

We present a linear program for allocating tasks to processors such that the maximum cluster utilization is met.

Constraints

We present the linear constraints that must be satisfied by the previous variables.

Number of requests. The first set of constraints ensures that no more than f_k tasks of application a_k are scheduled:

$$1 \leq k \leq q,\ 0 \leq \sum_{i=1}^{m} \delta_i^k \leq f_k \tag{6.1}$$

This equation also enforces the limits on the δ_i^k values: $1 \leq i \leq m,\ 1 \leq k \leq q,\ 0 \leq \delta_i^k \leq f_k$.

Bounded computing capacity. Each processor cannot serve more computing power than what it offers on the period T:

$$1 \leq i \leq m,\ \sum_{k=1}^{q} d_k.\delta_i^k \leq w'_i \tag{6.2}$$

Applications available on a processor. If applications are only available on some processors, *i.e.,* $E_i^k = 1$, then we need to restrict the mapping of the tasks on this subset of processors. E_i^k being a parameter of the problem, a constant. The next set of constraints enforces δ_i^k to 0 if application a_k is not present on processor i:

$$1 \leq i \leq m, 1 \leq k \leq q, \delta_i^k \leq E_i^k.f_k \tag{6.3}$$

Maximum cluster utilization. In each cluster, the percentage of computing power used must not be higher than β_j. At least, this constraint cannot be violated due to our applications assignment, but a cluster could already be "overloaded" due to the other tasks already present in the Gantt of this cluster. The utilization percentage is given by the following equation for a given cluster C_j:

$$\frac{\sum_{\{P_i\ :\ P_i \in C_j\}} \left(\alpha_i.w_i + \sum_{k=1}^{q} d_k.\delta_i^k\right)}{\sum_{\{P_i\ :\ P_i \in C_j\}} w_i} \leq \beta_j \tag{6.4}$$

Thus, the maximum utilization constraint can be expressed as follows:

$$1 \leq j \leq p,\ \sum_{\{P_i\ :\ P_i \in C_j\}} \sum_{k=1}^{q} d_k.\delta_i^k \leq \max\left\{0;\ \sum_{\{P_i\ :\ P_i \in C_j\}} w_i\left(\beta_j - \alpha_i\right)\right\} \tag{6.5}$$

The maximum is here to prevent negative upper bounds: if the cluster is already overloaded due to the other tasks scheduled on it, then the $\sum_{\{P_i\ :\ P_i \in C_j\}} w_i\left(\beta_j - \alpha_i\right)$ value is negative. Note that even though the constraint uses a maximum, it remains linear, as its right hand side is in fact a constant.

Objective

Our goal is to schedule as many tasks per application as possible, and to have almost the same ratio of scheduled to released applications, $\frac{\sum_{i=1}^{k} \delta_i^k}{f_k}$, for all applications. Thus, we aim at having $\frac{\sum_{i=1}^{k} \delta_i^k}{f_k}$ as close as possible to 1 for each application, so as to serve as many requests as possible.

So we aim at minimizing:

$$obj = \max_{1 \le k \le q} \left\{ 1 - \frac{\sum_{i=1}^{k} \delta_i^k}{f_k} \right\} \tag{6.6}$$

This reduces to minimizing μ, where μ is defined as follows:

$$1 \le k \le q, \ 1 - \frac{\sum_{i=1}^{k} \delta_i^k}{f_k} \le \mu \tag{6.7}$$

Linear program

We now present a linear program, $\mathcal{LP}_6$, for scheduling tasks on processors under the maximum cluster utilization constraints, when the real slacks' sizes are not taken into account.

Minimize μ
Subject to constraints (6.1), (6.2), (6.3), (6.5), and (6.7). $(\mathcal{LP}_6)$

Note that such a formulation of the program only guaranties that the maximum of $1 - \frac{\sum_{i=1}^{k} \delta_i^k}{f_k}$ on the applications will be minimized. Which may possibly leave unscheduled tasks, whereas some scheduling possibilities are still present. In order to maximize the number of scheduled tasks for each application, we can recursively solve $(\mathcal{LP}_6)$, find which application $a_{k'}$ has the lowest $1 - \frac{\sum_{i=1}^{k'} \delta_i^{k'}}{f_{k'}}$ value, then fix its variables to the values $(\mathcal{LP}_6)$ has found, and run it again without this application in the problem: $a_{k'}$ now becomes part of the Gantt, and for the other applications, we fix the lower bound on δ_i^k to the δ_i^k found in the previous execution of $(\mathcal{LP}_6)$.

Before moving on to describing how tasks can be scheduled at the server level, we present how new constraints on memory usage can easily be added to the model.

Adding memory constraints

As already stated, we consider that all required data is already present on the platform. Another way of seeing this, is that before the T period starts, the data is sent to whatever relevant processor, *i.e.,* processor where an application is scheduled. Thus, we slightly modify the problem. We no longer require that E_i^k be specified statically (E_i^k is now a variable of the problem: $E_i^k \in \{0, 1\}$), instead, we consider that each application a_k requires a certain amount $size_k$ of data be present on the processor where it runs. Two kinds of models can comply with this new set of constraints. Either the data is present directly at the server level, or it is present at the cluster level in a shared repository (for example an NFS partition). Of course, the available memory is on a machine is limited by the amount of disk storage available. We present two sets of constraints for these two model:

Data present on the processors. We first consider that data is stored at the processor level, each processor P_i has a limit on the available memory: M_{P_i}. Hence, the data placement constraints, and thus the applications placement constraints can be expressed as follows:

$$1 \le i \le m, \ \sum_{k=1}^{q} size_k . E_i^k \le M_{P_i} \tag{6.8}$$

Constraint (6.8) can be added to Linear Program ($\mathcal{LP}_6$) to form ($\mathcal{LP}_7$):

$$\begin{array}{l}\textbf{Minimize } \mu \\ \textbf{Subject to constraints } (6.1), (6.2), (6.3), (6.5), (6.7), \text{ and } (6.8).\end{array} \tag{$\mathcal{LP}_7$}$$

Application present on the clusters. Another way of seeing the problem, is to have a shared data repository on each cluster. If an application's data is present on a cluster's repository, then processors of this cluster can execute this application. As previously, we consider that we have a memory constraint on the clusters' repositories: M_{C_j}, which denotes the maximum disk space on the C_j cluster's repository. Finally, we declare $R_j^k \in \{0,1\}$, $R_j^k = 1$ if the data for a_k is present on cluster C_j, $R_j^k = 0$ otherwise. We have the following new sets of constraints:

$$1 \leq j \leq p, \ \sum_{k=1}^{q} size_k . R_j^k \leq M_{C_j} \tag{6.9}$$

and

$$1 \leq j \leq p, \ \forall P_i \in C_j, \ 1 \leq k \leq q, \ E_i^k = R_j^k \tag{6.10}$$

Constraints (6.9) and (6.10) can be added to Linear Program ($\mathcal{LP}_6$) to form ($\mathcal{LP}_8$):

$$\begin{array}{l}\textbf{Minimize } \mu \\ \textbf{Subject to constraints } (6.1), (6.2), (6.3), (6.5), (6.7), (6.9), \text{ and } (6.10).\end{array} \tag{$\mathcal{LP}_8$}$$

6.2.2 Scheduling Requests at the Server Level

Using one of the presented linear program, we are now able to obtain a "valid" placement of the tasks on the processors such that the maximum cluster utilization constraint is respected. This mapping of tasks onto processors is given by the δ_i^k variables: whenever their value is equal to 0, we know that no task of application a_k have to be scheduled on processor P_i, and whenever its value equals x we can assign "at most" x tasks of a_k on P_i. We explain thereafter why we say "at most."

Problem with the obtained assignment

The next step is to effectively schedule the tasks at the processors level. If we were using a divisible load model (applications with checkpoint-restart), then there would be no problem, as any valid placement would lead to a valid scheduling: any application could run in as many partial execution as required to finish the computation. For example, consider Figure 6.3a, it presents the initial state of the cluster, and a mapping of tasks on the processors. Now, if we consider this placement, and a divisible load model, then we would end-up with a valid schedule which would be given by Figure 6.3b: as each task can be split into as many subtasks as required, we can easily fit the tasks into the available slacks. In this case, we would not, in fact, be exactly in the divisible load model, as if we were in the divisible load model, a task would not have to be executed entirely on one processor, it could be split upon several processors. Note that if we were using the divisible load model, the linear programs presented above could be simplified, as δ_i^k could be defined as a real value, and not as an integer value, which would directly give us a valid schedule, and what more would simplify the resolution of linear program ($\mathcal{LP}_6$).

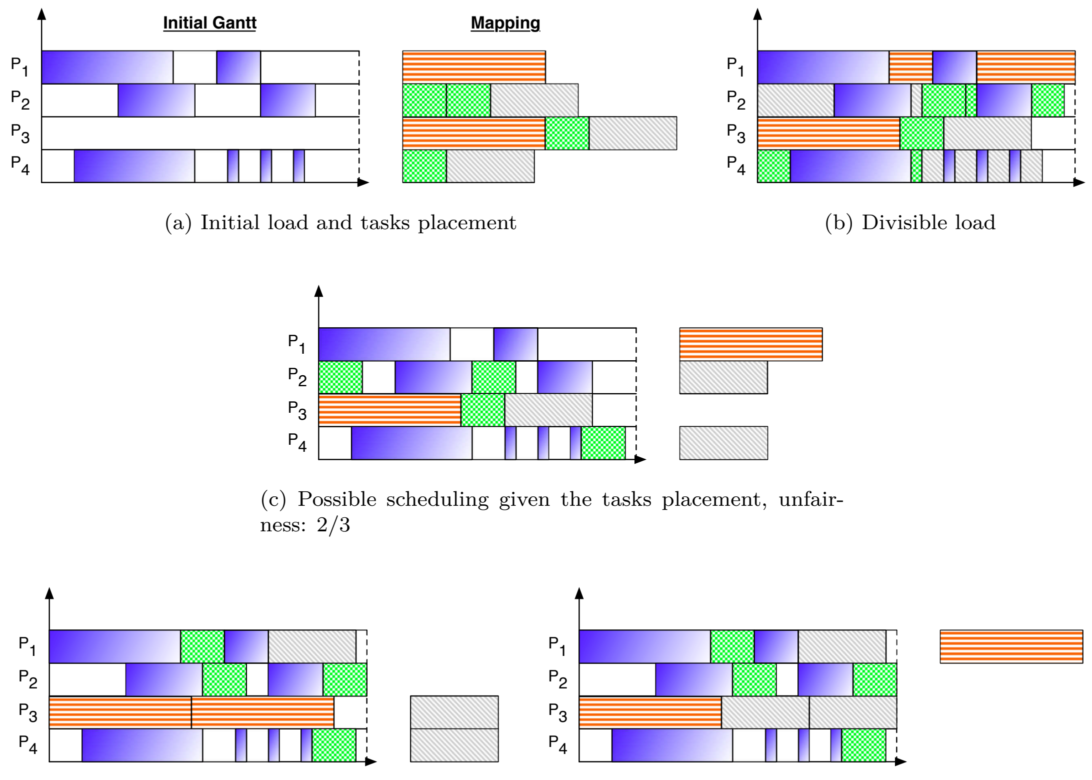
However this model does not reflect how most of the applications effectively work: checkpoint-restart mechanisms are not present in lots of applications, and dividing the work into small chunks is not always possible. Hence, if we consider the applications as a monolithic bloc of work that cannot be divided, we then need to fit each task exactly within a slack. Figures 6.3c, 6.3d, and 6.3e show possible schedules when:

- Figure 6.3c: respecting the given tasks allocations,
- Figures 6.3d and 6.3e: allowing changes in tasks allocation, *i.e.,* migration of tasks between processors.

As explained in the previous section, we use as "fairness" metric the ratio of scheduled tasks to requested tasks per bag-of-tasks, which as to have almost the same value for all bags-of-tasks. Thus, we can define the *unfairness* of a schedule as the difference between the lowest and the highest of these ratios:

$$unfairness = \min_{1 \le k \le q} \left\{ 1 - \frac{\sum_{i=1}^{k} \delta_i^{k'}}{f_k} \right\} - \max_{1 \le k \le q} \left\{ 1 - \frac{\sum_{i=1}^{k} \delta_i^{k'}}{f_k} \right\} \tag{6.11}$$

Where $\delta_i^{k'}$ is the number of effectively, and correctly scheduled tasks at the server level. The unfairness is nothing more than the difference of the percentage of scheduled tasks between the application having the highest percentage, and the application having the lowest one. As can be seen, finding a valid and efficient schedule might not be easy given the allocation obtained using the linear programs ($\mathcal{LP}_6$), ($\mathcal{LP}_7$) or ($\mathcal{LP}_8$). In some cases it might even prove impossible. Consider Figure 6.3c, its unfairness is 2/3. Even though the allocation given in Figure 6.3d gives the same unfairness, it manages to schedule one more task. Finally, the optimal schedule in this case is given in Figure 6.3e, which does not respect the initial assignment.



(a) Initial load and tasks placement

(b) Divisible load

(c) Possible scheduling given the tasks placement, unfairness: 2/3

(d) Scheduling 1, unfairness: 2/3

(e) Scheduling 2, unfairness: 1/2

Figure 6.3: Comparison between divisible load and scheduling with atomic tasks.

Thus, we need strategies to maximize the number of really allocated tasks, and reduce the unfairness. We present two sets of constraints to deal with the above mentioned problem.

Adding constraints to the initial *lp*

We can add constraints to the initial linear program, ($\mathcal{LP}_6$) so as to take into account slacks in the Gantt of each processor and only schedule tasks that can fit in, and thus obtain a valid schedule. However, adding new constraints to a linear program increases the problem size, and hence the problem resolution time. The first set of constraints does not provide a valid and optimal solution, but merely guides the solution towards potentially a good processor. It may requires further processing to obtain a valid schedule. The second set of constraints provides an exact solution, and a valid schedule. However, this latter solution adds lots of new variables and constraints.

Maximum slack. We can had constraints on the maximum task size a processor can process, *i.e.,* the maximum workload that can be processed in a slack. Let s_i be the maximum slack size for processor P_i, *i.e.,* the maximum task size that can be scheduled on this processor. We have the following constraints:

$$1 \leq i \leq m,\ 1 \leq k \leq q,\ E_i^k.d_k \leq s_i \tag{6.12}$$

Adding these constraints does not really solve the problem, as this only ensures that there is at least one slack where we can schedule a task, this does not tell anything concerning the other slacks. Thus, we still need to schedule tasks correctly at processor level. However, this guides the algorithms, as it ensures that on a given processor, each task, taken independently, can be scheduled on at least one slack.

Real slacks. We can take into account all slacks' sizes. This approach, even if it directly provides a valid schedule, greatly increases the problem size. For each processor P_i, let ρ_i be the number of slacks on P_i, for $1 \leq t \leq \rho_i$ let s_i^t be the "size" of slack t, *i.e.,* the computing power offered by the slack. Thus, we have another way of defining w_i' based on the slacks: $w_i' = \sum_{t=1}^{\rho_i} s_i^t$.
The δ_i^k variables are no longer sufficient, we need to add the following integer variables: $\delta_{i,t}^k \in [0, f_k]$. Similarly to δ_i^k, $\delta_{i,t}^k$ represents the number of tasks of applications a_k scheduled on slack t of processor P_i.
Thus we have the following constraints:

$$1 \leq i \leq m,\ 1 \leq t \leq \rho_i,\quad \sum_{k=1}^{q} d_k.\delta_{i,t}^k \leq s_i^t \tag{6.13}$$

$$1 \leq i \leq m,\ 1 \leq k \leq q,\quad \sum_{t=1}^{\rho_i} \delta_{i,t}^k = \delta_i^k \tag{6.14}$$

Equation (6.13) states that a slack cannot compute more work than what it offers, and Equation (6.14) only computes the total number of instances of application a_k on processor P_i. The fact that $\delta_{i,t}^k$ must be in the range $[0, f_k]$ is taken care of by Equation (6.1). Given these new sets of constraints, the linear program becomes:

Minimize μ
Subject to constraints (6.1), (6.3), (6.5), (6.7), (6.10), (6.13), and (6.14). $(\mathcal{LP}_9)$

Remark. *The problem, when dealt with the real slacks constraints (Equations* (6.13)*, and* (6.14)*), is in fact similar to considering that each slack is an individual processor. Indeed, we hide the Gantt of each processor when we try to directly map the applications on the slacks.*

6.3 Taking Into Account Release Dates

In the previous section, we dealt with the problem of scheduling bags of tasks under the maximum cluster utilization constraint, when all tasks were released at the same time. The goal of this schedule was to map as many tasks as possible per bag of tasks on a given period, while having "fairness" between the bags of tasks. What happens now if we consider that the tasks can arrive at anytime? We consider that within a bag of tasks, each task can have a distinct release date.

In this section, we consider the scheduling of bags of tasks under the maximum cluster utilization constraint, when tasks are released at different dates. We do not use the same objective as before, but instead we rely on the stretch. As previously, we will iteratively build the solution.

6.3.1 Problem Formulation without Maximum Cluster Utilization Constraint

Prior to presenting the problem with the full set of constraints, we first give a linear program formulation for the problem of scheduling bags of tasks with release dates, such that all tasks have a stretch no bigger than a given maximum stretch $\mathcal{S}$.

Problem statement

We rely on the same system assumptions presented in Section 6.1, apart from the fact that now, each task has a *release date* r_k^g: let $1 \leq k \leq q,\ 1 \leq g \leq f_k,\ r_k^g$ be the release date for instance g of task a_k.

Now, suppose that we computed beforehand the optimal makespan $MS_{k,g}^*$ of scheduling instance g of a_k alone on the platform, *i.e.,* it is the only job to be scheduled on the platform as it currently is (with the already scheduled tasks, the initial Gantt). Also, let $\mathcal{S}$ be the maximum stretch we want to attain, *i.e.,* we aim at scheduling all tasks such that none of them has a stretch higher than $\mathcal{S}$. Finally, let $MS_{k,g}$ be the makespan obtained by instance g of a_k with our scheduling strategy. $MS_{k,g}$ has to respect the following constraint:

$$\begin{aligned} & 1 \leq k \leq q,\ 1 \leq g \leq f_k, \quad \frac{MS_{k,g}}{MS_{k,g}^*} \leq \mathcal{S} \\ \iff \quad & 1 \leq k \leq q,\ 1 \leq g \leq f_k, \quad r_k^g + MS_{k,g} \leq r_k^g + \mathcal{S} \times MS_{k,g}^* \end{aligned}$$

Given the above formulae, we are able to define the maximum completion time for instance g of a_k, such that it respects the maximum stretch $\mathcal{S}$, *i.e.,* its deadline. The deadline for a task is given by the following formula:

$$c_k^g = r_k^g + \mathcal{S} \times MS_{k,g}^* \tag{6.15}$$

If a task cannot complete no later than its deadline, then the maximum stretch cannot be attained, and consequently we will need to try a larger maximum stretch.

As we consider tasks' release dates, we also need to know the starting time and the ending time of each slack on the processors. Let b_i^t be the starting time of slack t on processor P_i. We do not introduce another variable for its ending time, as it is equal to $b_i^t + s_i^t$.

Once a maximum stretch $\mathcal{S}$ has been chosen, we divide the total execution time into time-intervals whose bounds are the tasks' release dates or deadlines, or the slacks' begin or ending dates. We define a set of epochal times: $\Gamma_l \in \Gamma^L = \bigcup_{k=1}^{q} \bigcup_{g=1}^{f_k} \{r_k^g\} \cup \bigcup_{k=1}^{q} \bigcup_{g=1}^{f_k} \{c_k^g\} \cup \bigcup_{i=1}^{n} \bigcup_{t=1}^{\rho_i} \{b_i^t\} \cup \bigcup_{i=1}^{n} \bigcup_{t=1}^{\rho_i} \{b_i^t + s_i^t\}$, such that $\Gamma_l \leq \Gamma_{l+1}$. We do not take into account dates Γ_l such that $\Gamma_l < \min\{r_k^g : 1 \leq k \leq q,\ 1 \leq g \leq f_k\}$, nor dates $\Gamma_{l'}$ such that $\Gamma_{l'} > \max\{c_k^g : 1 \leq k \leq q,\ 1 \leq g \leq f_k\}$. As some dates may be equal, we can end-up with empty intervals. However, for the sake of simplicity, we do not remove these empty intervals. We denote by $L = \left|\Gamma^L\right|$ the number of dates.

The idea behind the algorithm presented thereafter consists in running a task g of applications a_k during its execution window $[r_k^g, c_k^g]$ on a single slack. However, as each slack can be divided by several time-intervals, we will determine which percentage of the task has to be executed on the time interval so as to fulfill the constraints.

Let $\lambda_i^{k,g}(\Gamma_l, \Gamma_{l+1})$ be a real variable representing the percentage of instance g of application a_k executed on processor P_i during epoch $[\Gamma_l, \Gamma_{l+1}]$. Note that we need these variables only on the periods where P_i has available computing power, *i.e.,* during the period where P_i does not already have scheduled tasks. Thus, we define new $\lambda_i^{k,g}(\Gamma_l, \Gamma_{l+1})$ variables only on the slacks of P_i, as outside these slacks, the processor has no available computing power, and hence $\lambda_i^{k,g}(\Gamma_l, \Gamma_{l+1}) = 0$ on such intervals.

We also need to determine for each slack, the percentage of instance g of application a_k executed on slack t of processor P_i during epoch $[\Gamma_l, \Gamma_{l+1}]$. Let $s_i^t(\Gamma_l, \Gamma_{l+1})$ be a real variable representing the available computing power offered by the slack t on processor P_i during the interval $[\Gamma_l, \Gamma_{l+1}]$ (hence s_i^t is just the sum of $s_i^t(\Gamma_l, \Gamma_{l+1})$ over all intervals $[\Gamma_l, \Gamma_{l+1}]$).

Given these new variables, we are now able to write a linear program for scheduling the instances on the processors, without maximum cluster utilization constraint.

Linear program formulation

Completion. Every instance of each application has to be computed entirely, *i.e.,* the sum of percentages $\lambda_i^{k,g}(\Gamma_l, \Gamma_{l+1})$ must be equal to 1. Setting this constraint exactly to 1 enforces the fact that an instance has to be computed. Thus, and if a single instance cannot be scheduled, then we the answer must that their is no solution to this problem with the given stretch $\mathcal{S}$.

$$1 \leq i \leq n,\ 1 \leq k \leq q,\ 1 \leq g \leq f_k,\ 1 \leq l \leq L-1,\ 0 \leq \lambda_i^{k,g}(\Gamma_l, \Gamma_{l+1}) \leq 1 \tag{6.16}$$

and

$$1 \leq k \leq q,\ 1 \leq g \leq f_k,\ \sum_{i=1}^{n} \sum_{\substack{l=1 \\ \Gamma_l \geq r_k^g \\ \Gamma_{l+1} \leq c_k^g}}^{L-1} \lambda_i^{k,g}(\Gamma_l, \Gamma_{l+1}) = 1 \tag{6.17}$$

Bounded computing capacity. The computing capacity of a node should not be exceeded on any time-interval. As noted above, we only deal with intervals where it is worth computing $\lambda_i^{k,g}(\Gamma_l, \Gamma_{l+1})$, *i.e.,* on the slacks.

$$\begin{gathered}1 \leq i \leq n,\ 1 \leq t \leq \rho_i,\ l \in \left\{x : 1 \leq x < L,\ b_i^t \leq \Gamma_x \leq \Gamma_{x+1} \leq b_i^t + s_i^t\right\} \\ \sum_{k=1}^{q} \sum_{g=1}^{f_k} \lambda_i^{k,g}(\Gamma_l, \Gamma_{l+1}) .d_k \leq s_i^t(\Gamma_l, \Gamma_{l+1})\end{gathered} \tag{6.18}$$

Atomic computation. Every instance has to be computed atomically, *i.e.,* its computation cannot take place nor on several processors, nor on several slacks. Let $\lambda_{i,t}^{k,g}$ be an Boolean variable representing the percentage of instance g of applications a_k computed on slack t of processor P_i.

$$1 \leq i \leq n,\ 1 \leq k \leq q,\ 1 \leq g \leq f_k,\ \lambda_{i,t}^{k,g} \in \{0,1\} \tag{6.19}$$

and

$$1 \leq i \leq n,\ 1 \leq t \leq \rho_i,\ 1 \leq k \leq q,\ 1 \leq g \leq f_k,\quad \sum_{\substack{l=1 \\ \Gamma_l \geq b_i^t \\ \Gamma_{l+1} \leq b_i^t + s_i^t}}^{L-1} \lambda_i^{k,g}(\Gamma_l, \Gamma_{l+1}) = \lambda_{i,t}^{k,g} \tag{6.20}$$

Along with Equation 6.17, these equations ensure that only one slack can compute a task.

Release and deadline. Each computation percentage for every instances g of application a_k should always be equal to 0 outside the interval $[r_k^g, c_k^g]$.

$$\begin{gathered}1 \leq k \leq q,\ 1 \leq g \leq f_k,\ 1 \leq i \leq n,\ 1 \leq t \leq \rho_i, \\ l \in \{x : 1 \leq x < L,\ 0 \leq \Gamma_x \leq \Gamma_{x+1} \leq r_k^g\},\quad \lambda_i^{k,g}(\Gamma_l, \Gamma_{l+1}) = 0\end{gathered} \tag{6.21}$$

and

$$\begin{array}{r}1 \leq k \leq q,\ 1 \leq g \leq f_k,\ 1 \leq i \leq n,\ 1 \leq t \leq \rho_i, \\ l \in \{x : 1 \leq x < L,\ c_k^g \leq \Gamma_x\},\quad \lambda_i^{k,g}(\Gamma_l, \Gamma_{l+1}) = 0\end{array} \tag{6.22}$$

Objective. The goal of this linear program is to find whether or not a valid schedule exists under the constraint that the maximum stretch equals $\mathcal{S}$. Thus only a valid schedule is necessary, and we can aim at, for example, minimizing the makespan of all applications. Hence, minimizing:

$$\max_{\substack{1 \leq i \leq n \\ 1 \leq t \leq \rho_i}} \left\{ b_i^t + s_i^t - \sum_{k=1}^{q} \sum_{g=1}^{f_k} \sum_{\substack{l = 1 \\ \Gamma_l \geq b_i^t \\ \Gamma_{l+1} \leq b_i^t + s_i^t}}^{L-1} \lambda_i^{k,g}(\Gamma_l, \Gamma_{l+1}).d_k \right\} \tag{6.23}$$

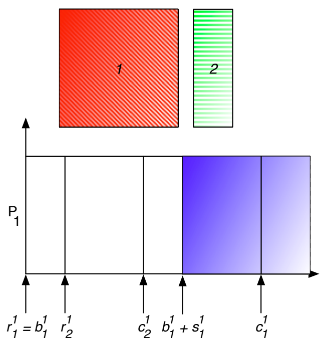
This reduces to minimizing ms, where ms is defined as follows:

$$1 \leq i \leq n,\ 1 \leq t \leq \rho_i,\ b_i^t + s_i^t - \sum_{k=1}^{q} \sum_{g=1}^{f_k} \sum_{\substack{l = 1 \\ \Gamma_l \geq b_i^t \\ \Gamma_{l+1} \leq b_i^t + s_i^t}}^{L-1} \lambda_i^{k,g}(\Gamma_l, \Gamma_{l+1}).d_k \leq ms \tag{6.24}$$

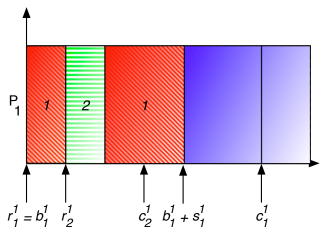
Linear program formulation. From the above sets of constraints, the problem can be formulated by a linear program, as follows:

Minimize ms
Subject to constraints (6.16), (6.17), (6.18), (6.19), (6.20), (6.21), (6.22) and (6.24). $(\mathcal{LP}_{10})$

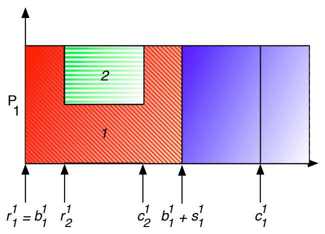
This problem formulation returns a solution where many jobs can run concurrently on a same processor. Figure 6.4 presents the sort of schedule we can obtain with this problem formulation. Figure 6.4a presents the initial conditions. We have two applications a_1 and a_2 with only one task to schedule for each. We aim at having a schedule with a maximum stretch $\mathcal{S} = 2$. Figure 6.4b presents a schedule respecting the values of $\lambda_{i,t}^{k,g}$ obtained by the linear program: a_1 is alone on the periods $\left[r_1^1, r_2^1\right]$ and $\left[c_2^1, v_1^1 + s_1^1\right]$, and a_1 and a_2 are both present on the period $\left[r_2^1, c_2^1\right]$. If no two tasks can run concurrently, then this formulation of the linear problem does not return in fact a valid schedule, as it would require a divisible load model. However, if two applications can run concurrently, then we can obtain a valid schedule, as shown in Figure 6.4c.



(a) Initial setting



(b) Schedule obtained with the $\lambda_{i,t}^{k,g}$: the computations may not be atomic



(c) Possible schedule with jobs running concurrently

Figure 6.4: Problem with linear program ($\mathcal{LP}_{10}$) formulation: schedule may not be atomic. This figure gives an example with two tasks, and $\mathcal{S} = 2$.

6.3.2 Taking into Account the Maximum Cluster Utilization

The maximum cluster utilization constraint states that in each cluster, the percentage of computing power used must not be higher than β_j on a given period T, at least not due to our applications assignment.

In Section 6.2, we considered that the period T (on which the maximum cluster utilization was verified) was fixed, and happened only once. We now consider a more general case, where we want to verify the constraint on several periods. So as to be really general, we will consider that these periods can overlap, and do not necessarily have the same size.

Each period i on which we want to check the maximum cluster utilization is defined by a beginning and an ending date: T_i^{begin} and T_i^{end}. We need to add these dates into Γ^L, and re-define all $\lambda_i^{k,g}(\Gamma_l, \Gamma_{l+1})$ and $s_i^t(\Gamma_l, \Gamma_{l+1})$ variables over this new Γ^L set. We also define a set containing all the periods on which we need to check the constraint: $\Gamma_{set}^T = \left\{ \left\{ T_1^{begin}, \ldots, T_1^{end} \right\}, \left\{ T_2^{begin}, \ldots, T_2^{end} \right\}, \ldots \right\}$. Each set $\left\{ T_i^{begin}, \ldots, T_i^{end} \right\}$ contains all dates of Γ^L included in the period $\left[T_i^{begin}, T_i^{end} \right]$.

For all $(\Gamma_l, \Gamma_{l+1}) \in \Gamma^L$ we define $w_i\left(\Gamma_l, \Gamma_{l+1}\right)$ to be the total computing power of processor P_i for time interval $[\Gamma_l, \Gamma_{l+1}]$. Also, let $\alpha_i\left(\Gamma_l, \Gamma_{l+1}\right)$ be the percentage of computing power of P_i used over that time interval by other applications, note that due to our definition of Γ^L, $\alpha_i\left(\Gamma_l, \Gamma_{l+1}\right)$ equals either 0 or 1 (0 if the period falls into a slack, and 1 otherwise).

As an example, we could define the maximum cluster utilization periods as follows. Let's consider a fixed period T, and consider that we aim at having the maximum cluster constraint verified "at anytime". Of course, this cannot exactly be done, so we consider that such a period starts anytime a task of the Gantt starts or end on the platform, or when a task is released or at its deadline.

Thus, we modify Γ^L so as to include the corresponding epochal times: starting from any date in $\Gamma_l \in \Gamma^L$ (as defined in Section 6.3.1), we add the epochal time $\Gamma_l + T$ if this date does not exist, and if $\Gamma_l + T < \max\left\{\Gamma_l \in \Gamma^L\right\}$. We also add the dates required to encompass the latest task deadline. Thus, we obtain the following set of periods on which we need to verify the constraint: $\Gamma^T_{set} = \{\{\Gamma_0, \dots \Gamma_0 + T\}, \{\Gamma_1, \dots \Gamma_1 + T\}, \dots\}$. As can be seen on this example, the definition of the periods can be quite complex, and several periods can overlap.

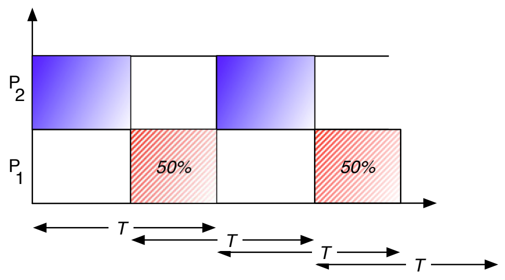
With these new periods definitions we are now able to give a set of constraints for checking the maximum cluster utilization, which can be added to Linear Program ($\mathcal{LP}_{10}$):

$$1 \leq j \leq p,\ \forall \Gamma^T \in \Gamma^T_{set}$$
$$\sum_{\{P_i\ :\ P_i \in C_j\}} \sum_{t=1}^{\rho_i} \sum_{\substack{l=1 \\ (\Gamma_l, \Gamma_{l+1}) \in \Gamma^T}}^{|\Gamma^T|-1} \sum_{k=1}^{q} \sum_{g=1}^{f_k} \lambda_i^{k,g}(\Gamma_l, \Gamma_{l+1}).d_k$$
$$\leq \max\left\{0, \sum_{\{P_i\ :\ P_i \in C_j\}} \sum_{\substack{l=1 \\ (\Gamma_l, \Gamma_{l+1}) \in \Gamma^T}}^{|\Gamma^T|-1} w_i(\Gamma_l, \Gamma_{l+1})(\beta_j - \alpha_i(\Gamma_l, \Gamma_{l+1}))\right\} \tag{6.25}$$

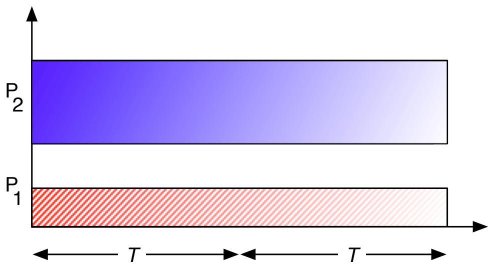
As previously given in Equation (6.5) we need to use the max function so as to deal with periods on which the maximum cluster utilization constraint is violated by the initial Gantt.

This new constraint does not help coping with the fact that we may need a divisible load model to obtain a valid schedule. Figure 6.5a presents an example where the linear program result would require a divisible load model. In this example we require that no more than half of the computing power is used. Tasks on P_2 are part of the initial Gantt. On P_1 we schedule a single task, which is split into two so as to respect the maximum utilization cluster on any T period.

It may also lead to a schedule where the computing power of nodes are only partially used. For example, on Figure 6.5b, we require than no more than 3/4 of the computing power is used. The task on P_2 is part of the initial Gantt. The linear program can return a solution where only 50% of P_1 is used during two T periods (in fact this is equivalent to having this task split into two, each sub-task using the whole computing power on twice less time).



(a) May require a divisible load model



(b) May lead to partially use a node

Figure 6.5: Problems with the maximum cluster utilization constraint.

Thus, adding the maximum cluster utilization constraint does not leverage the problem of atomic computations, and even worse, it can lead the linear program to a solution requiring a divisible load model, or an execution model where we are allowed to use partially the processors.

Remark (Maximum cluster utilization constraint formulation)**.** *Even though the given example relies on periods of size T starting at each available date in the original Gantt and at each date induced by the tasks' arriving dates and deadlines to compute the maximum utilization constraint, any kind of periods* *can be considered. The periods do not necessarily have the same size, nor do they need to overlap, or even cover the whole Gantt, there can be gaps between the periods.*

Moreover, we considered that the periods are present on all clusters, which leads to a very constrained problem. In fact, the formulation of the periods on which are computed the maximum utilization constraints can be changed. Any kind of period can be used, for as long as all processors of a cluster are subjected to the same periods. Indeed, adapting the problem to other kinds of periods is quite straightforward, as one would need to modify the Γ^T_{set} *and* Γ^L *to adapt to her problem. The formulation with different periods between clusters only requires to define sets of periods per cluster. Doing so would in fact lower the number of variables and constraints in our problem, but as the formulation is already complex, adding new indices to the variables depending on the cluster on which they are checked would have lowered the readability of the equations.*

6.3.3 Coping with the Atomic Tasks Problem

We will now extend Linear Program ($\mathcal{LP}_{10}$) in order obtain a valid schedule that would not require a divisible load model. We need to check two constraints so as to enforce the atomic tasks model. First, we need to make sure that any task is entirely executed on contiguous periods. Secondly, we need to make sure that anytime a task spans on several periods, the *inner* periods are entirely used, otherwise this would mean that we could slow down the computer and use only partially its computing power.

Contiguous period computations

We first deal with the fact that tasks need to be computed atomically. We will add new constraints to our problem. Let's first define new variables:

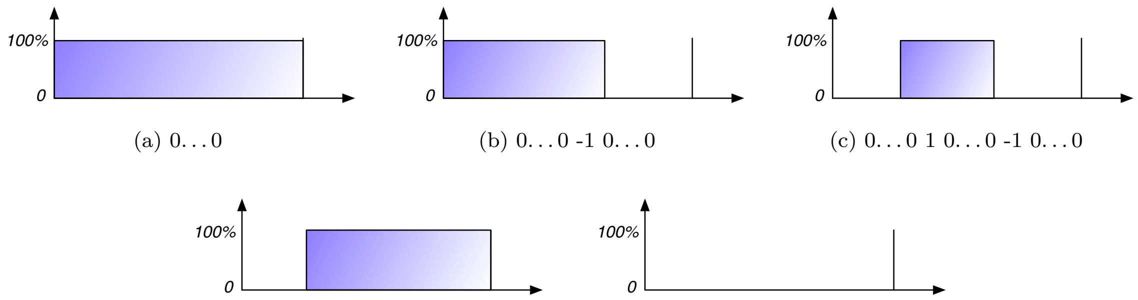
– let $y_i^{k,g}(\Gamma_l, \Gamma_{l+1}) \in \{0, 1\}$ be the period placement variable for instance g of application a_k, on processor P_i during period (Γ_l, Γ_{l+1}). It equals 1 if and only if $\lambda_i^{k,g}(\Gamma_l, \Gamma_{l+1}) > 0$, *i.e.*, if and only if part of the task is mapped on P_i during the period (Γ_l, Γ_{l+1}).
– also let $z_i^{k,g}(\Gamma_l, \Gamma_{l+2})$ be the difference between $y_i^{k,g}(\Gamma_{l+1}, \Gamma_{l+2})$ and $y_i^{k,g}(\Gamma_l, \Gamma_{l+1})$:

$$z_i^{k,g}(\Gamma_l, \Gamma_{l+2}) = y_i^{k,g}(\Gamma_{l+1}, \Gamma_{l+2}) - y_i^{k,g}(\Gamma_l, \Gamma_{l+1}) \in \{-1, 0, 1\}$$

What is the meaning of these $z_i^{k,g}(\Gamma_l, \Gamma_{l+2})$ variables? They will help us determine the valid and invalid schedules. We aim at having a real "atomic computation" for all instances, and prevent the case presented on Figure 6.5a. Thus, we need to ensure that on a given slack, an instance is not mapped on two periods which are not consecutive. Let's analyze the behavior of the $z_i^{k,g}(\Gamma_l, \Gamma_{l+2})$ variables on valid and invalid schedules.

Figure 6.6 presents the only valid placements for a task on a slack, the sub-figures' titles give the corresponding values taken by the $z_i^{k,g}(\Gamma_l, \Gamma_{l+2})$ variables:

– Figure 6.6a: the task uses the entire slack, all $z_i^{k,g}(\Gamma_l, \Gamma_{l+2})$ variables equal 0.
– Figure 6.6b: the task is scheduled at the beginning of the slack, and finishes before the end of the slack, all $z_i^{k,g}(\Gamma_l, \Gamma_{l+2})$ variables equal 0 apart from the one when the task ends, which equals -1.
– Figure 6.6c: the task starts during the slack, and ends before the end of the slack, all $z_i^{k,g}(\Gamma_l, \Gamma_{l+2})$ variables equal 0 apart from the ones when the task starts and ends, which respectively equal 1 and -1.
– Figure 6.6d: the task starts during the slack, and ends when the slack ends, all $z_i^{k,g}(\Gamma_l, \Gamma_{l+2})$ variables equal 0 apart from the one when the task starts, which equals 1.
– Figure 6.6e: the task is not scheduled on this slack, all $z_i^{k,g}(\Gamma_l, \Gamma_{l+2})$ variables equal 0.

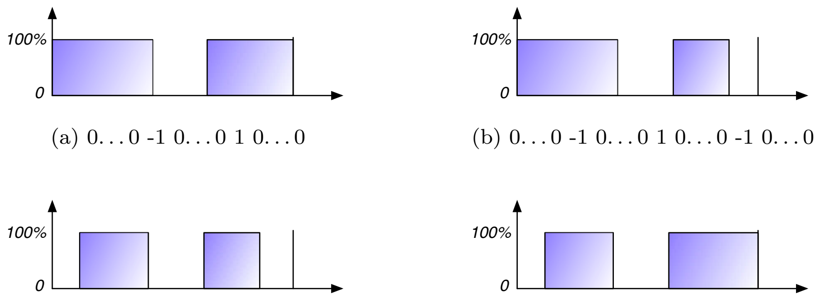


(a) 0. . . 0 (b) 0. . . 0 -1 0. . . 0 (c) 0. . . 0 1 0. . . 0 -1 0. . . 0

(d) 0. . . 0 1 0. . . 0 (e) 0. . . 0

Figure 6.6: Allowed cases.

Similarly, Figure 6.7 presents examples of invalid placements: the task cannot be split into several sub-tasks. Note that in this figure, we only present the cases where the task is split into two. The cases where the task is split into more than two is just a matter of adding part of the task into the remaining empty periods. The "$z_i^{k,g}(\Gamma_l, \Gamma_{l+2})$ patterns" would not be much affected per se.



(a) 0. . . 0 -1 0. . . 0 1 0. . . 0 (b) 0. . . 0 -1 0. . . 0 1 0. . . 0 -1 0. . . 0

(c) 0. . . 0 1 0. . . 0 -1 0. . . 0 1 0. . . 0 -1 0. . . 0 (d) 0. . . 0 1 0. . . 0 -1 0. . . 0 1 0. . . 0

Figure 6.7: Some of the forbidden cases.

The $z_i^{k,g}(\Gamma_l, \Gamma_{l+2})$ variables can take three different values:
- 0: when nothing changes during two consecutive periods $[\Gamma_l, \Gamma_{l+1}]$ and $[\Gamma_{l+1}, \Gamma_{l+2}]$, *i.e.,* either the task does not execute on these periods, or it does, but not both,
- 1: the task does not execute on the first period $[\Gamma_l, \Gamma_{l+1}]$, and executes on the second $[\Gamma_{l+1}, \Gamma_{l+2}]$,
- -1: the task execute on the first period $[\Gamma_l, \Gamma_{l+1}]$, and does not on the second $[\Gamma_{l+1}, \Gamma_{l+2}]$.

Analyzing the valid and invalid schedules, and the corresponding values of the $z_i^{k,g}(\Gamma_l, \Gamma_{l+2})$ variables, we can see emerging patterns of allowed and forbidden combinations of $z_i^{k,g}(\Gamma_l, \Gamma_{l+2})$ variables. A schedule is valid on a given slack if on this slack we do not have one of the following patterns on the $z_i^{k,g}(\Gamma_l, \Gamma_{l+2})$ variables:

1. a -1 value is followed by a 1 value at some point,
2. we cannot have more than one 1 value, nor can we have several -1 values.

With these observations, we are now able to define four new sets of constraints:

$$\begin{array}{l} 1 \le i \le n,\ 1 \le t \le \rho_i, 1 \le k \le q,\ 1 \le g \le f_k, \\ l \in \{x : 1 \le x < L \ \wedge\ b_i^t \le \Gamma_x \le \Gamma_{x+1} \le b_i^t + s_i^t\}, \end{array} \quad \lambda_i^{k,g}(\Gamma_l, \Gamma_{l+1}) \le y_i^{k,g}(\Gamma_l, \Gamma_{l+1}) \tag{6.26}$$

$$\begin{array}{l} 1 \leq i \leq n,\ 1 \leq t \leq \rho_i,\ 1 \leq k \leq q,\ 1 \leq g \leq f_k, \\ l \in \{x : 1 \leq x < x+2 \leq L\ \wedge\ b_i^t \leq \Gamma_x \leq \Gamma_{x+1} \leq \Gamma_{x+2} \leq b_i^t + s_i^t\}, \\ \qquad y_i^{k,g}(\Gamma_{l+1}, \Gamma_{l+2}) - y_i^{k,g}(\Gamma_l \Gamma_{l+1}) = z_i^{k,g}(\Gamma_l, \Gamma_{l+2}) \end{array} \tag{6.27}$$

$$\begin{array}{l} 1 \leq i \leq n,\ 1 \leq t \leq \rho_i,\ 1 \leq k \leq q,\ 1 \leq g \leq f_k, \\ l, l' \in \{x : 1 \leq x < x+2 \leq L\ \wedge\ b_i^t \leq \Gamma_x \leq \Gamma_{x+2} \leq b_i^t + s_i^t\},\ \textit{and } l < l' \\ \qquad -1 \leq z_i^{k,g}(\Gamma_{l'}, \Gamma_{l'+2}) + z_i^{k,g}(\Gamma_l, \Gamma_{l+2}) \leq 1 \end{array} \tag{6.28}$$

$$\begin{array}{l} 1 \leq i \leq n,\ 1 \leq t \leq \rho_i,\ 1 \leq k \leq q,\ 1 \leq g \leq f_k, \\ l' = x \text{ such that } b_i^t = \Gamma_x, \\ l \in \{x : 1 \leq x < x+2 \leq L\ \wedge\ b_i^t \leq \Gamma_x \leq \Gamma_{x+2} \leq b_i^t + s_i^t\}, \\ \qquad -1 \leq y_i^{k,g}(\Gamma_{l'}, \Gamma_{l'+1}) + z_i^{k,g}(\Gamma_l, \Gamma_{l+2}) \leq 1 \end{array} \tag{6.29}$$

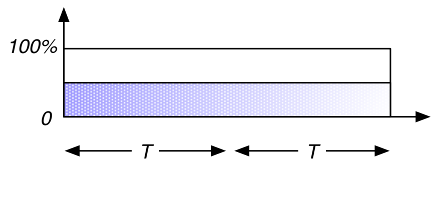
Let's now explain these new equations. Variables $y_i^{k,g}(\Gamma_l, \Gamma_{l+1})$ precise whether or not instance g of application a_k is mapped on period $[\Gamma_l, \Gamma_{l+1}]$: if $y_i^{k,g}(\Gamma_l, \Gamma_{l+1})$ equals 1, then thanks to Equation (6.26) we know that at least part of the instance is computed on this period, otherwise it equals 0.

Equation (6.27) defines a function with values in $\{-1, 0, 1\}$, *i.e.*, the $z_i^{k,g}(\Gamma_l, \Gamma_{l+2})$ variables. Equation (6.28) prevents this function from having two 1 or two -1 values on the whole slack. As already stated, a value of 1 means that previously the instance was not mapped on the slack, and that now a part of it is mapped. A value of -1 means that previously the instance was mapped, and that it is not anymore on the considered period. Thus, if we want to ensure that the instance is computed atomically we need to ensure that we cannot have no two 1 or -1 values for the same instance on the slack, hence Equation (6.28).

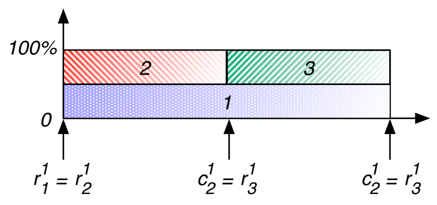
Equation (6.29) is quite similar to Equation (6.27). The difference is that we add a fictitious variable $y_i^{k,g}(\Gamma_l, \Gamma_l)$ equal to 0 at the beginning of the slack, so as to have in this case a value of $z_i^{k,g}(\Gamma_l, \Gamma_{l+2})$ equal to 1 (as the instance starts at the beginning of the slack). This constraint is here to cope with the particular case presented in Figure 6.7a. In this case, Equation (6.27) cannot detect that we have an invalid configuration, as we only have a -1 and a 1 value, and thus from the point of view of Equation (6.27) this case is similar to the one presented in Figure 6.6c.

Computing power utilization

Having dealt with the atomic computation problem, two problems still remain. We want to make sure that whenever a processor is used, its computing power is entirely used, and that a valid schedule without no two tasks running concurrently at the same time can be found. Figure 6.8 sums up these two problems.



(a) Computing power partially utilized



(b) No valid schedule can be extracted from the linear program without scheduling tasks concurrently

Figure 6.8: Problems with the computing power utilization.

To cope with the problem of computation power partially used, we need to add news constraints stating that the computation on a given period of a slack has to fulfill certain conditions:

Case 1 whenever at least a task do not start or end in the considered period, then the slack has to be completely used (the sum of the computations done on this period has to be equal to the offered computing power). Otherwise this would mean that we can slow down the computation of our tasks on a given period. See Figure 6.9c, period 3.

Case 2 if a task is either starting or finishing on the considered period, then the slack can be partially used. This does not require to slow down computation, it only means that if several tasks fit in this period, they can be scheduled one after another on the given period while meeting the requirements of the maximum cluster utilization. See Figures 6.9a and 6.9b periods 1 to 5, and Figure 6.9c, periods 1, 2, 4, 5.

Case 3 Any other possibility is a combination of Cases 1 and 2.

The problem of concurrent executing tasks will be taken care of when dealing with the first problem.

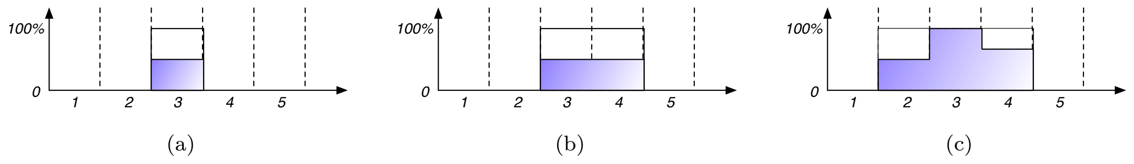


Figure 6.9: Allowed computation patterns on a processor.

On a period $[\Gamma_l, \Gamma_{l+1}]$, we need to constrain the utilization of a processor to $s_i^t(\Gamma_l, \Gamma_{l+1})$ if we fall into Case 1. Note that whenever a slack is composed of only one or two periods, then we do not need to check these constraints, as the utilization can always be a fraction of the total available computation power, *i.e.,* we always fall into Case 2. The same holds for any period at the beginning or the end of a slack.

We need a mean of forcing the sum of the works computed on P_i during $[\Gamma_l, \Gamma_{l+1}]$ to be equal to $s_i^t(\Gamma_l, \Gamma_{l+1})$, whenever required. In order to cope with the concurrent execution problem, we also need to decide if a task has to entirely use the computing power of a processor on a given period. Thus, let $\eta_i^{k,g}(\Gamma_l \Gamma_{l+1})$ be a Boolean variable equal to 1 if and only if instance g of application k has to use the whole available computing power of P_i during $[\Gamma_l, \Gamma_{l+1}]$. A task has to use the whole computing power on a period if it is neither starting, nor finishing on this period. As an example, on Figure 6.9c, on period 1, 2, 4, and 5, $\eta_i^{k,g}(\Gamma_l \Gamma_{l+1})$ should equal 0, and on period 3 it should equal 1. In fact, $\eta_i^{k,g}(\Gamma_l \Gamma_{l+1})$ variables do not need to be Boolean variables, but only real variables with values in $[0,1]$:

$$1 \leq i \leq n,\ 1 \leq k \leq q,\ 1 \leq g \leq f_k,\ \Gamma_l, \Gamma_{l+1} \in \Gamma^L, \Gamma_l \leq \Gamma_{l+1},\ 0 \leq \eta_i^{k,g}(\Gamma_l \Gamma_{l+1}) \leq 1 \tag{6.30}$$

$\eta_i^{k,g}(\Gamma_l \Gamma_{l+1})$ variables are set to 0 on periods outside slacks. We now add constraints on $\eta_i^{k,g}(\Gamma_l \Gamma_{l+1})$ so as to give them their correct $\{0,1\}$ values:

$$1 \leq i \leq n,\ 1 \leq t \leq \rho_i,\ 1 \leq k \leq q,\ 1 \leq g \leq f_k,$$
$$l \in \left\{x : 1 \leq x-1 < x+2 \leq L\ \wedge\ b_i^t \leq \Gamma_{x-1} \leq \Gamma_x \leq \Gamma_{x+1} \leq \Gamma_{x+2} \leq b_i^t + s_i^t\right\},$$
$$\eta_i^{k,g}(\Gamma_l \Gamma_{l+1}) \geq y_i^{k,g}(\Gamma_{l-1}, \Gamma_l) + y_i^{k,g}(\Gamma_l, \Gamma_{l+1}) + y_i^{k,g}(\Gamma_{l+1}, \Gamma_{l+2}) - 2 \tag{6.31}$$
$$\leq y_i^{k,g}(\Gamma_l, \Gamma_{l+1}) \tag{6.32}$$
$$\leq y_i^{k,g}(\Gamma_{l-1}, \Gamma_l) \tag{6.33}$$
$$\leq y_i^{k,g}(\Gamma_{l+1}, \Gamma_{l+2}) \tag{6.34}$$

Equation (6.31) forces the value of $\eta_i^{k,g}(\Gamma_l \Gamma_{l+1})$ to 1 when the application is present on this period $[\Gamma_l \Gamma_{l+1}]$, and it is neither starting or terminating. In this case, $y_i^{k,g}(\Gamma_{l-1}, \Gamma_l)$, $y_i^{k,g}(\Gamma_l, \Gamma_{l+1})$ and $y_i^{k,g}(\Gamma_{l+1}, \Gamma_{l+2})$ all equal 1, and the sum minus two equals 1. In all the other possible configurations the sum is lower than 3, and this equation does not force $\eta_i^{k,g}(\Gamma_l \Gamma_{l+1})$ to 1. Equations (6.32) to (6.34) forces the value to 0 when:

– Equation (6.32): a_k^g is not present on period $[\Gamma_l \Gamma_{l+1}]$
– Equation (6.33): a_k^g is not present on the previous period, *i.e.,* $[\Gamma_{l-1}\Gamma_l]$
– Equation (6.34): a_k^g is not present on the following period, *i.e.,* $[\Gamma_{l+1}\Gamma_{l+2}]$

Finally, we can add the constraint on the minimum computing power utilization on any period for any processor:

$$\begin{aligned}&1 \leq i \leq n,\ 1 \leq t \leq \rho_i,\ 1 \leq k \leq q,\ 1 \leq g \leq f_k\\ &l \in \left\{x : 1 \leq x < x+1 \leq L \ \wedge\ b_i^t \leq \Gamma_x \leq \Gamma_{x+1} \leq b_i^t + s_i^t\right\},\\ &\qquad \lambda_i^{k,g}\left(\Gamma_l, \Gamma_{l+1}\right).d_k \geq s_i^t\left(\Gamma_l, \Gamma_{l+1}\right).\eta_i^{k,g}\left(\Gamma_l, \Gamma_{l+1}\right)\end{aligned} \tag{6.35}$$

Complete linear program formulation

Using these constraints, and the definition of the maximum cluster utilization given by Equation (6.25), we can now ensure that the solution of the linear program will give us a solution to our problem.

In fact, we do not really need to minimize the maximum makespan, as we only wish to ensure that all the tasks have a stretch lower than or equal to $\mathcal{S}$. Hence, we only need to find a feasible solution with maximum stretch $\mathcal{S}$, *i.e.,* verify that our set of constraints accept at least one solution. In practice, to verify that this is the case, we use a fictitious linear objective function to our set of constraints. Thus, we can use as objective function, for example, the sum of all $\lambda_{i,t}^{k,g}$, that we try to maximize. When all tasks are mapped, this sum equals the total number of tasks submitted to the system. Linear Program ($\mathcal{LP}_{11}$) gives the complete sets of constraints that need to be verified to solve the problem.

$$\begin{array}{ll}\textbf{Maximize} \displaystyle\sum_{i=1}^{m}\sum_{t=1}^{\rho_i}\sum_{k=1}^{q}\sum_{g=1}^{f_k}\lambda_{i,t}^{k,g} & \\ \textbf{Subject to constraints} & (6.16), (6.17), (6.18), (6.19), (6.20), (6.21),\\ & (6.22), (6.24), (6.25), (6.26), (6.27), (6.28),\\ & (6.30), (6.29), (6.31), (6.32), (6.33), (6.34), \text{and } (6.35).\end{array} \tag{$\mathcal{LP}_{11}$}$$

Remark. *Why do we use such a complex representation for our problem? Indeed, one could wonder why not use a simpler representation that would directly use tasks' release date and deadline, with as an objective function the minimization of the maximum stretch on all tasks. This formulation would indeed lead to less variables and constraints for the simplified problem without the constraint on the maximum utilization (Section 6.3.1), and would moreover give us directly the schedule. However, using this representation, we cannot add the maximum cluster utilization constraint, as this would lead to computing intervals' overlap between tasks' schedule and the periods on which we check the utilization constraint. Computing these overlaps leads to computing minimum and maximums, which are not linear functions.*

Note, that other constraints such as memory constraints presented in Section 6.2.1 could easily be added to the formulation of Linear Program ($\mathcal{LP}_{11}$).

6.3.4 Obtaining a Real Schedule

Linear Program ($\mathcal{LP}_{11}$) provides a valid mapping of the tasks, such as a schedule with maximum stretch $\mathcal{S}$, and respecting the maximum cluster utilization constraint can be found. Even though this formulation does not provides us with the direct starting and ending time of each task, the $\lambda_i^{k,g}\left(\Gamma_l, \Gamma_{l+1}\right)$ variables give the amount of computing power that is required for each task on each period. Thanks to constraints (6.26) to (6.35), we know that we can find a schedule where no two tasks will have to run concurrently.

Obtaining a valid schedule can be done as follows:

1. First of all, we need to determine the starting and ending dates of all tasks that are scheduled on three or more than three periods. Apart from the periods where the task starts or ends, we are sure that the task uses the whole "inner" periods (see Figure 6.9c). Thus, for these tasks, we

can easily determine the starting and ending dates. If instance g of a_k starts during $[\Gamma_l, \Gamma_{l+1}]$ on P_i, then its starting time will be $\Gamma_{l+1} - \frac{\lambda_i^{k,g}(\Gamma_l,\Gamma_{l+1}).d_k}{w_i(\Gamma_l,\Gamma_{l+1})} \times (\Gamma_{l+1} - \Gamma_l)$. Similarly, if it ends during $(\Gamma_{l'}, \Gamma_{l'+1})$, then its ending time will be $\Gamma_{l'} + \frac{\lambda_i^{k,g}(\Gamma_{l'},\Gamma_{l'+1}).d_k}{w_i(\Gamma_{l'},\Gamma_{l'+1})} \times (\Gamma_{l'+1} - \Gamma_{l'})$.

2. Once all the heavily constrained tasks have been dealt with, we can move on to the remaining tasks. All the tasks that remain are scheduled either on one or two periods (see Figures 6.9a and 6.9b). We can schedule them in a greedy fashion. We consider each processor independently. Starting from the first period, we consider each period one after another. On the considered period, first schedule as soon as possible all tasks that execute exactly during this period, in any order. Then, schedule the remaining tasks (those which span on two periods) as soon as possible. Move on to the next period and repeat this process.

Using this greedy approach, we build a valid schedule. Note that a task which spans on two periods may, in the end, not execute on these two periods, but only on the first of the second one. This does not matter, and does not invalidate the constraints, as we ensure that the execution remains during the time frame of these two periods.

6.3.5 Finding the Best Maximum Stretch

To find the best attainable stretch, we perform a binary search on the possible $\mathcal{S}$ values. Each time a value for $\mathcal{S}$ is considered, we try to find a valid solution to ($\mathcal{LP}_{11}$). Three values are required for the binary search. The lower bound on the achievable stretch is 1. We also need an initial upper bound on the achievable stretch, $\mathcal{S}_{max}$. We can use a naive approach to define $\mathcal{S}_{max}$. For example, we could wait for the last task to be released before starting to schedule the tasks. Then start to schedule the latest tasks first, and finish with the first release task, by using for example *Minimum Completion Time* (MCT) under the maximum utilization constraint. This ensures that we have a valid schedule, and we can take the maximum stretch on all instances for $\mathcal{S}_{max}$.

Finally, we need the termination criterion of the binary search, that is the gap between two possible stretches: ϵ. We suppose that the determination of the value of ϵ is out of the scope of this research, as it may depend on the users' wishes. So we consider that ϵ is provided by the users. Suppose that we are given $\epsilon > 0$, then the determination of $\mathcal{S}$ using a binary search can be realized using Algorithm 6.1.

Algorithm 6.1 Binary search

```
1.1:  $\mathcal{S}_{inf} \leftarrow 1$
1.2:  $\mathcal{S}_{sup} \leftarrow \mathcal{S}_{max}$
1.3:  while $\mathcal{S}_{sup} - \mathcal{S}_{inf} > \epsilon$ do
1.4:      $\mathcal{S} \leftarrow \frac{\mathcal{S}_{sup} - \mathcal{S}_{inf}}{2}$
1.5:      if Linear Program ($\mathcal{LP}_{11}$) has no solution then
1.6:          $\mathcal{S}_{inf} \leftarrow \mathcal{S}$
1.7:      else
1.8:          $\mathcal{S}_{sup} \leftarrow \mathcal{S}$
1.9:      end if
1.10: end while
1.11: return $\mathcal{S}_{sup}$
```

At each step, the research interval is divided by two. Thus, we need $O\left(\log \frac{\mathcal{S}_{mac}}{\epsilon}\right)$ steps to find the interval containing the optimal stretch $\mathcal{S}_{opt}$. At the end, $\mathcal{S}_{opt} \in [\mathcal{S}_{inf}, \mathcal{S}_{sup}]$, with $\mathcal{S}_{sup} - \mathcal{S}_{inf} \leq \epsilon$, such that $\mathcal{S}_{sup} \leq \mathcal{S}_{opt} + \epsilon$, and $\mathcal{S}_{sup}$ is achievable.

An interesting and powerful approach is proposed in [122] and [143] to find the optimal stretch on scheduling problems under stretch minimization objective. It relies on the definition of stretch-intervals. Within each stretch-interval the ordering of the dates of Γ^L is the same, whatever the value of $\mathcal{S}$. The idea is then to perform a binary search on the research intervals, and to try to minimize the stretch of all tasks with the linear program (the objective function is then the minimization of $\mathcal{S}$). Unfortunately, this approach cannot be applied in our case due to Equation (6.35) which becomes quadratic, as the dates Γ_l vary linearly with $\mathcal{S}$.

6.4 Conclusion

In this chapter, we have presented linear program solutions for the problem of offline scheduling of bags-of-tasks on a platform where the resources utilization could be bounded on given periods of time. We iteratively presented the solutions, and showed how to circumvent the problems inherent to basic linear programming formulations. Two cases are considered. The simplest one considers that all tasks are released at the same time, and only one period T, on which access to resources is limited, is considered. In this case the objective is to schedule as many tasks as possible from each bag-of-tasks. Then, we considered the case where tasks can arrive at anytime. The solution derives from the solution for the problem without release dates. Complex and interleaved limitation periods are considered, we give linear constraints leading to a correct tasks' mapping onto the processors. The objective in this case, is to minimize the maximum stretch, which is attained using a binary search.

Scheduling problems are strongly related to dynamic middleware adaptation and re-deployment. Based on execution history, we can check that the metric we want to optimize has an acceptable value, and if not, we might need to adapt the number of machines controlled by the middleware. Thus, the next step in middleware deployment would be to have a feedback mechanism that would send information about scheduling results to the middleware deployment controller, which in turn would make adequate decisions on whether the middleware deployment should be adapted or not.

Part IV

Eyes in the Sky

Chapter 7

Contribution to Cosmological Simulations: Implementation of a Distributed Platform

Deploying the middleware on sets of "close" nodes, and scheduling jobs on available resources is only one part of the problem of efficiently running cosmological simulations on a grid. A few needs remain. Most of them can be addressed by the middleware with a few modifications, such as data management, and scheduling the tasks of a workflow (though information specific to cosmological simulations is needed); others have to be directly solved. For example, when deploying the middleware across several platforms, communication problems can appear due to several security policies; and users still require a graphical interface that hides the complexity of submitting jobs to the middleware.

In this chapter [1], we deal with the problem of effectively running cosmological simulations on a grid of computers in a transparent way. This involves managing communications, scheduling tasks, managing tasks' dependencies in cosmological simulations' workflows, and providing a comprehensible user interface. Thus, the purpose of this chapter is twofold:

1. Our first concern is to provide a transparent and easy access to computing resources to the users. As those are not necessarily versed into the "art" of using a command line, even less to use a distributed environment, we propose an end-to-end infrastructure. Our solution provides an easy and comprehensible web interface to submit cosmological simulations, coupled with the Diet middleware to access resources. The communication layer complexity can also be hidden by using PadicoTM in order to cope with firewall and network issues. We illustrate the motivation of this work with the deployment of this whole infrastructure over three computing elements: Grid'5000, a local cluster GdsDmi and our computational and web server Graal, the last two elements reside in Lyon, France.
2. Secondly, we study the case of the deployment of cosmological applications on a grid. The goal of these applications is to simulate the evolution of particles, in order to study the formation of galaxies. The interest of these applications, relies in that they mix sequential and parallel codes, and that they can be assembled into a complex workflow. We present cosmological software modelization, and our Diet client/server implementation, along with experiments results.

The rest of the chapter is organized as follows. We first present our prototype architecture which provides transparent access to resources in Section 7.1. Then, we provide background on cosmological simulations and motivate the usage of simulations in this research field in Section 7.2. Finally, we present our work on cosmological simulations in Section 7.3.

1. The work presented in this chapter has been published in international conferences [ABB+06, CCC+07, DCD+09], and international journal [ABC+08].

7.1 Transparently Using a Grid

As already seen in the previous parts of the dissertation, using a grid is not straightforward. This is true when running a single sequential application, but even more when running complex scientific simulations. Those often require the use of parallel software, coupled with several other software. Such a sequence of applications is called a workflow, and is represented by a graph of tasks. The execution order of these tasks is defined by the control and data dependencies. The graph, *i.e.,* the assembly of tasks, can become quite complex, and executing it requires to manage both the jobs execution, and the data transfers and replications. Thus the need to have a workflow control manager, which automatically handles these issues. We present in this section, an end-to-end solution to run complex scientific applications on a grid.

7.1.1 Applications Management

As presented in the first part of this dissertation, one of the means to access distributed resources is to use a Grid-RPC middleware such as Diet. Managing the execution of applications on a grid is not only limited to being able to access resources, but it also means being able to move data around, and coordinate tasks so as to execute complex workflows.

The workflow paradigm on grids is well adapted for representing interdependent tasks, and the development of several workflow engines (DAGMan [2], GridAnt [31], MA$_{DAG}$ [ABB^{+}06], Pegasus [148], Taverna [133]) illustrates significant and growing interest in workflow management. The most commonly used model to represent a workflow is the graph, and especially the Directed Acyclic Graph (DAG). A DAG does not provide any control mechanisms (such as loops or if statements), but is generally sufficient to describe a large range of applications.

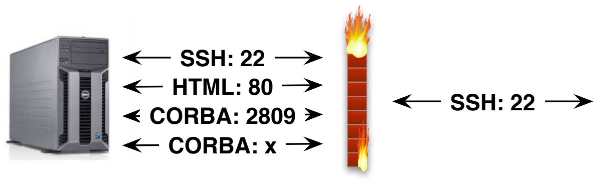
Diet introduces a new kind of agent: the MA$_{DAG}$, which is connected to the MA, as can be seen Figure 7.3. Instead of submitting requests directly to the MA, a client can submit a workflow to the MA$_{DAG}$, *i.e.,* an XML file containing the whole workflow description. The MA$_{DAG}$ will then take care of the workflow execution, and schedule it along with all the other workflows present in the system. The system can manage multiple workflows concurrently. Thus, the client only needs to describe the workflow in an XML format, then feed in the input data, and finally retrieve the output data when the workflow has finished its execution.

As data needs to be moved between consecutive tasks, we need an efficient data management tool. Diet can use Dagda [60] (Data Arrangement for Grid and Distributed Application), which allows data explicit or implicit replications and advanced data management on the grid such as data backup and restoration, persistency, and data replacement algorithm. A Dagda component is attached to each Diet element and follows the same hierarchical distribution. However, whereas Diet elements can only communicate following the hierarchy order (these communications appear when searching a service, and responding to a request), Dagda components will use the tree to find data, but once the data is found, direct communications will be made between the owner of the data and the element that requested it. The Dagda component associates an ID to each stored data, manages the transfers by choosing the "best" data source according to statistics about the previous transfers time. Just like Diet, Dagda uses the CORBA interface for inter-nodes communications.

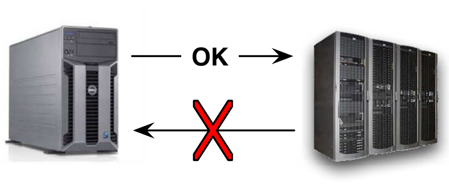
All in all, our resource management system is based on Diet which includes a workflow management system with the MA$_{DAG}$, and data management system with Dagda. The whole infrastructure relies on transparent communications with CORBA. However, communications can only be transparent when no firewall or network accessibility problems prevent CORBA from using whichever ports it needs.

7.1.2 Bypassing Firewalls

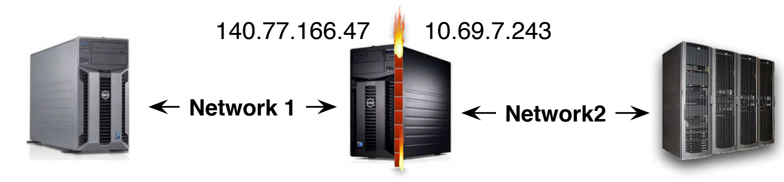
A recurrent problem when dealing with distributed resources is: how can we jointly use several computing resources that are managed by different entities? This often implies that even if you have an account on two different clusters, due to firewall rules and network design, you won't be able to communicate between the two clusters: only a few ports can be opened (usually only ssh is open), connections might not be bidirectional, or machines may be hidden behind a gateway which in turn does not reply with the same address depending on the network we're in. Figure 7.1 presents these different problems. A classical solution consists in manually creating ssh tunnels and encapsulating any required connection within these tunnels. However, if this solution may seem reasonable for a few clusters and ports, it quickly becomes prohibitively complicated, especially when running applications that dynamically open ports, such as DIET due to CORBA communications.



(a) Restricted open ports



(b) Unidirectional connections

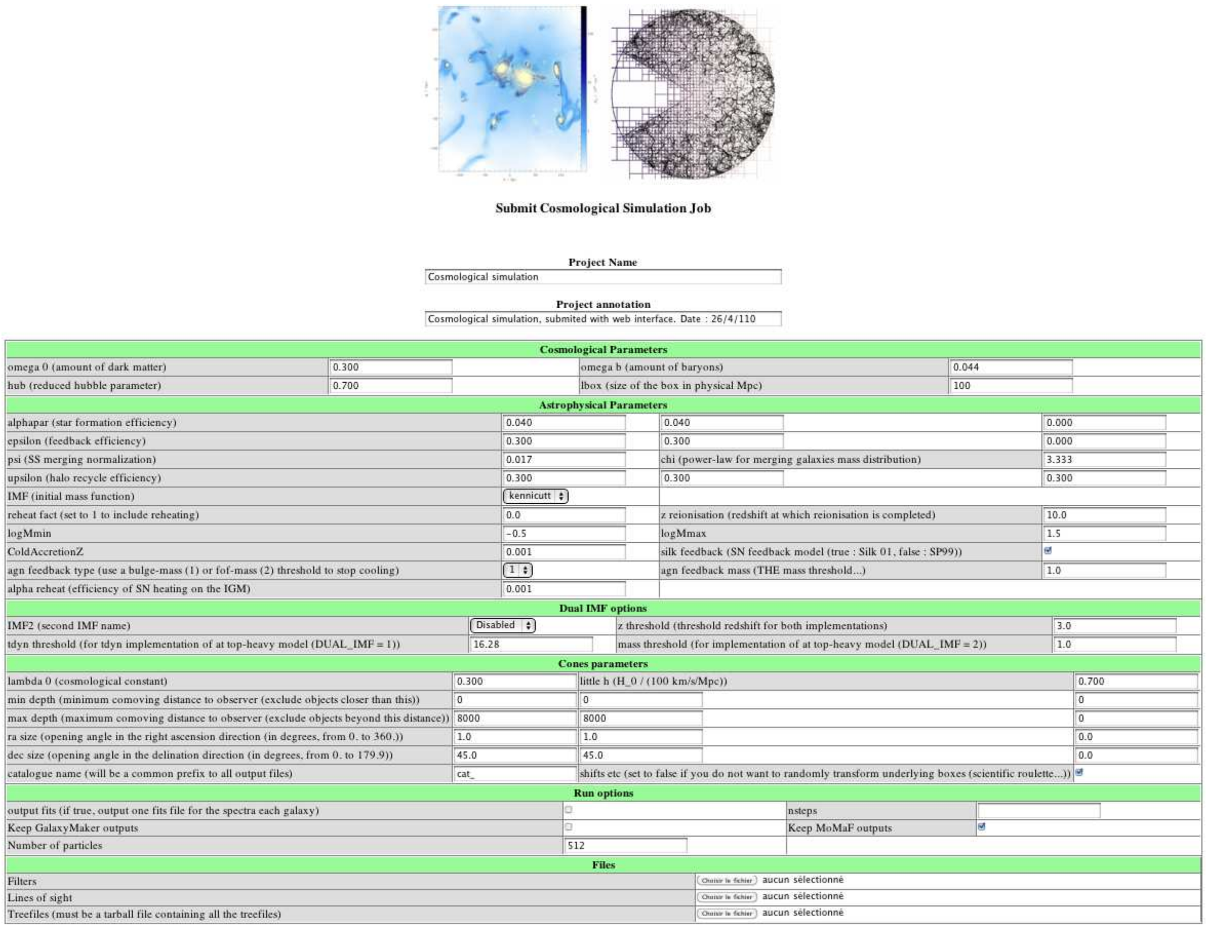


(c) Network "hidden" by a gateway

Figure 7.1: Connection problems within a distributed platform.

Including networking management within DIET would be complicated and quite limiting. Thus, we'd rather rely on an external tool capable of handling communications, and routing them in a suitable manner whenever required. To this end, we use PadicoTM [75]. PadicoTM aims at providing high performance communications and threads. Given a synthetic platform description with communication rules, PadicoTM is able to catch all communication routines within a program, and can apply routing, ssh tunneling, *etc.* Thus, running on top of PadicoTM, DIET is able to run across several clusters which only have for example the ssh port opened. Moreover, those clusters do not necessarily need to be able to communicate by ssh in a bidirectional manner. As presented on the example architecture figure, Figure 7.3, the only prerequisite is that the network is connected, PadicoTM then takes care of routing communications. Another big advantage of using PadicoTM, is that no code modification has to be made to use it: PadicoTM uses LD_PRELOAD in order to load its libraries in place of the standard libraries, before the program is executed. It then catches all communication system calls, and route them appropriately whenever required.

Using PadicoTM and its routing and tunneling facilities, we are now able to interconnect several platforms, even if nothing more than ssh is available between the sites. Finally, when all communication problems have been dealt with, a last obstacle remains: making the system accessible to non expert users.



Submit Cosmological Simulation Job

Project Name
Cosmological simulation

Project annotation
Cosmological simulation, submited with web interface. Date : 26/4/110

Cosmological Parameters			
omega 0 (amount of dark matter)	0.300	omega b (amount of baryons)	0.044
hub (reduced hubble parameter)	0.700	lbox (size of the box in physical Mpc)	100
Astrophysical Parameters			
alphapar (star formation efficiency)	0.040	0.040	0.000
epsilon (feedback efficiency)	0.300	0.300	0.000
psi (SS merging normalization)	0.017	chi (power-law for merging galaxies mass distribution)	3.333
upsilon (halo recycle efficiency)	0.300	0.300	0.300
IMF (initial mass function)	kennicutt		
reheat fact (set to 1 to include reheating)	0.0	z reionisation (redshift at which reionisation is completed)	10.0
logMmin	-0.5	logMmax	1.5
ColdAccretionZ	0.001	silk feedback (SN feedback model (true : Silk 01, false : SP99))	✓
agn feedback type (use a bulge-mass (1) or fof-mass (2) threshold to stop cooling)	1	agn feedback mass (THE mass threshold...)	1.0
alpha reheat (efficiency of SN heating on the IGM)	0.001		
Dual IMF options			
IMF2 (second IMF name)	Disabled	z threshold (threshold redshift for both implementations)	3.0
tdyn threshold (for tdyn implementation of at top-heavy model (DUAL_IMF = 1))	16.28	mass threshold (for implementation of at top-heavy model (DUAL_IMF = 2))	1.0
Cones parameters			
lambda 0 (cosmological constant)	0.300	little h (H_0 / (100 km/s/Mpc))	0.700
min depth (minimum comoving distance to observer (exclude objects closer than this))	0	0	0
max depth (maximum comoving distance to observer (exclude objects beyond this distance))	8000	8000	0
ra size (opening angle in the right ascension direction (in degrees, from 0. to 360.))	1.0	1.0	0.0
dec size (opening angle in the delination direction (in degrees, from 0. to 179.9))	45.0	45.0	0.0
catalogue name (will be a common prefix to all output files)	cat_	shifts etc (set to false if you do not want to randomly transform underlying boxes (scientific roulette...))	✓
Run options			
output fits (if true, output one fits file for the spectra each galaxy)	☐	nsteps	
Keep GalaxyMaker outputs	☐	Keep MoMaF outputs	✓
Number of particles	512		
Files			
Filters	Choisir le fichier aucun sélectionné		
Lines of sight	Choisir le fichier aucun sélectionné		
Treefiles (must be a tarball file containing all the treefiles)	Choisir le fichier aucun sélectionné		

Figure 7.2: Cosmological simulation submission web page.

7.1.3 Friendly User Interface

Though Diet hides the complexity of the grid to the end-user, and PadicoTM the complexity of the network, classical Diet client programs need to be executed using the command line. Moreover, this often implies that Diet and the client program be installed on the end-user computer. All this can be an obstacle for efficiently and easily running already complex applications. Fortunately, Diet can be interfaced with a web page to submit jobs. We developed a web page [2] based on the Diet-WebBoard [1] for cosmological simulations (we describe those simulations in the next section). It provides the following features:

- **Security**: only registered users can access the submission web site. Connection is made using a login/password method.
- **Job submission**: different web pages offer the possibility to submit jobs to different kind of cosmological applications: Ramses, MoMaF or GalaxyMaker job. The interface also provides an easy way of submitting parameter sweep jobs for the post-processing parts: the user specifies, for each parameter, the range of values she wishes to test for both GalaxyMaker and MoMaF, and the system creates the corresponding workflow and executes it. If for some reason a job fails, the system takes care of resubmitting it automatically after a fixed period, this until the job finishes properly, or is canceled by the user.
- **Data management**: once a job has finished, result data is sent back to the server as a tarball, it

2. The web page can be found at `http://graal.ens-lyon.fr:5544`. A test account is available: user=test and password=test. This account is restricted to page navigation, and cannot submit jobs.

is then made available on the web site. The user can leave it on the server, or decide to remove it.
- **Statistics**: on jobs execution time, on the number of jobs submitted over a given period, on the output size…
- **Diet management**: the Diet-WebBoard can also be in charge of deploying the Diet platform using GoDiet, and of controlling that the hierarchy is still working and re-deploying it when necessary.

Figure 7.2 presents a submission web page for cosmological simulations.

7.1.4 Overall Architecture

We sum up in Figure 7.3 the whole architecture that has been developed to run scientific applications on a grid environment, and more specifically cosmological simulations. This figure only presents the prototype architecture used to run simulations across two platforms, namely GdsDmi and Grid'5000. However this architecture could easily be extended to larger platforms. For as long as we have a connected network, PadicoTM can take care about the communication layer, and Diet the jobs management.

Figure 7.3 shows the following. The platform is constituted of three elements: two computing platforms, Grid'5000 and GdsDmi, and one web and computational server, Graal. No ports apart from port 22 (the ssh port) is opened, and only Graal can open direct connections towards GdsDmi and Grid'5000. No outgoing connection can be made from within Grid'5000, and no direct connection can be made from GdsDmi to Grid'5000. On top of the physical infrastructure, PadicoTM opens relevant tunnels between the three sites, thus rending the deployment of Diet possible. Diet is then deployed on top of PadicoTM with the MA, MA$_{DAG}$ and a few SeDs on Graal, and the rest of the hierarchy (LA and SeDs) is spread on Grid'5000 and GdsDmi. Finally, the web site is deployed on Graal, it launches the relevant clients whenever a job request is made.

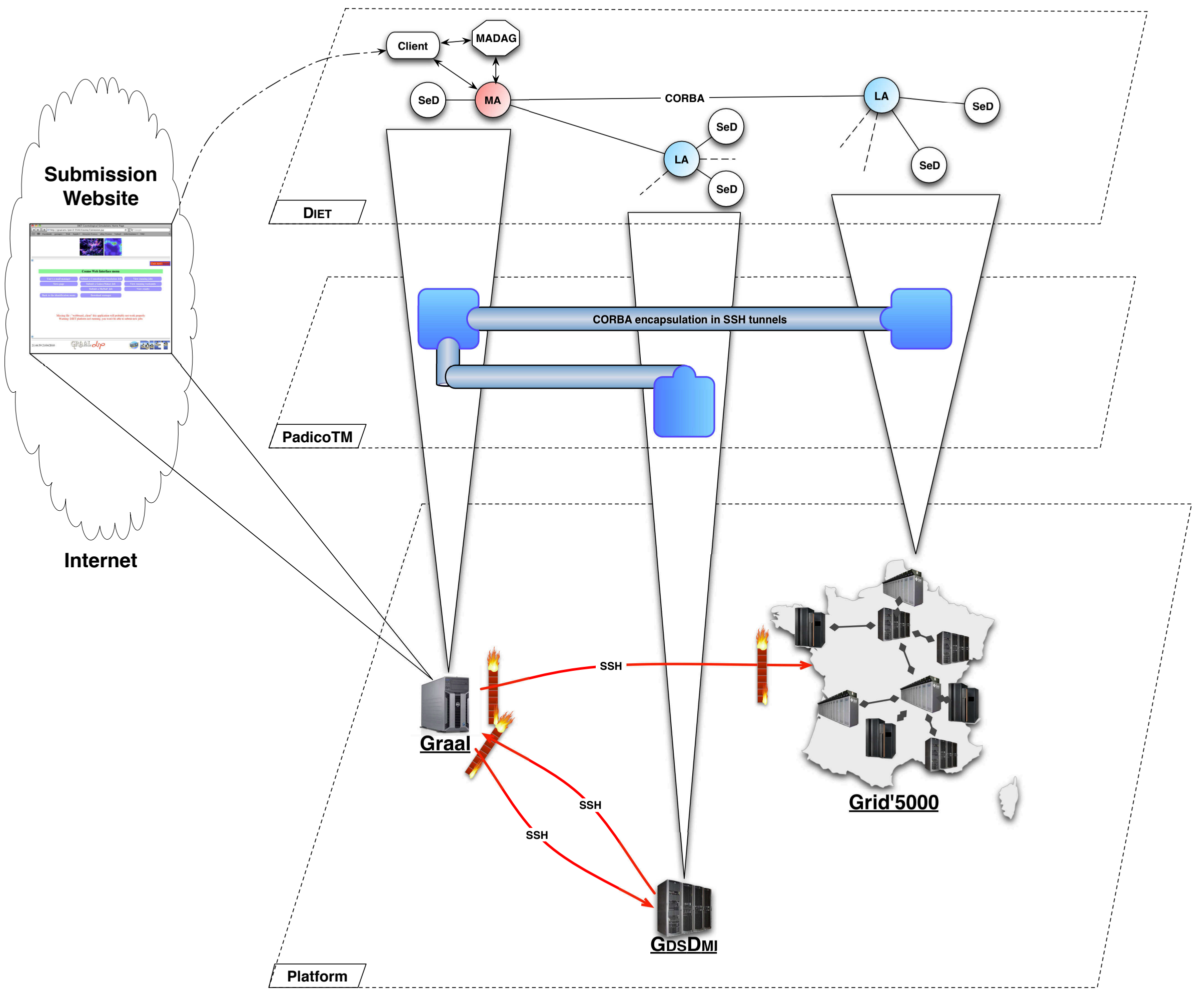


Figure 7.3: Overall architecture for running scientific applications on the grid.

7.2 Cosmological Simulations

Studying the formation of our universe can be decomposed into two orthogonal, but yet complementary domains: observations and simulations. It amounts to determining the position, composition, mass and velocity of galaxies, and thus creating catalogs of such known galaxies [136]. The process of studying the formation of the universe can be divided into five points:

1. Using real observational data, the physicists determine the matter density and velocity fields. From these, they can create catalogs of galaxies, containing their velocity, luminosity and mass.
2. Given observational data, numerical Initial Conditions (IC) are created for the simulations. They contain the position, velocity and mass of every particles in the universe just after the Big Bang.
3. *Dark matter* simulations are performed using a numerical simulator such as Gadget [151] or Ramses [160]. These applications produce only the distribution of matter and dark energy, but do not form the galaxies.
4. In order to compare the simulation results with the observational data (which only contains information about *baryonic matter*), we need to create visible matter, and thus galaxies, out of the simulation results. This is done by using software such as GaLICS [106] and MoMaF [50]. They create mock galaxies catalogs.
5. The last step consists in comparing the mock catalogs to the real catalogs, determining if the simulations fit the observations, and finally, using the simulations to determine parameters that cannot be seen by using only observations. Those results being in turn used to prepare future observations in order to better understand the physical processes at play.

Results obtained in step 5 are really important as they can be used to derive observation strategies, or even used in the design of new instruments:

– observation strategies: if the statistics on the simulations have a lot of fluctuations, it may be best to do ten observations in different part of the universe to have a good statistical representation of the universe; whereas if the statistics on the simulations are quite uniform, it may be best to do ten observations connected in a mosaic, in the same part of the universe, to have a greater angular field of view.
– new instruments: when designing a new observation instrument, we need to define for what kind of observations it has to be tuned. Depending on what we want to observe, it will need to be more or less sensitive: if the simulations state that the part of the universe that is studied is quite empty, then we'll need a very sensitive instrument that will be able to work with only a few photons, whereas if the part of the universe is full of galaxies, the instrument will receive more information.

Hence, once point 5 is done, we can go back to point 1 and start once again the whole process. We will now describe in details the five above-mentioned steps.

7.2.1 Determining Motions in the Universe

In 1964 Arno Penzias and Robert Wilson [140] discovered what is called the *Cosmic Microwave Background* (CMB). At the beginning the universe was dense and hot, filled with hydrogen plasma and radiations, and thus, was opaque. As it expanded, it cooled, became transparent, or at least seen with classical telescopes. However, when looking through a radio-telescope, we can observe isotropic radiations being roughly 2.7 ± 1 Kelvin (K). This is the CMB. Figure 7.4 presents the CMB as detected by WMAP (Wilkinson Microwave Anisotropy Probe). In fact some anisotropy can be observed when looking into details. One peculiar, and important fluctuation is a dipole anisotropy due to the peculiar motion of our local group of galaxies [90], *i.e.,* the CMB appears slightly warmer in the direction of the movement than in the opposite direction (*blueshift* and *redshift* motions due to the *Doppler shift*). Hence the need to study the peculiar motion of our *Local Group*, and determine the position, distance and velocity of galaxies.

When looking at the universe, we can detect regions of high density of matter, whereas some regions are almost empty of any matter (voids of the universe). If you could have a look at the universe "from afar", large structures of matter would appear. Indeed, the distribution of matter is not random, and

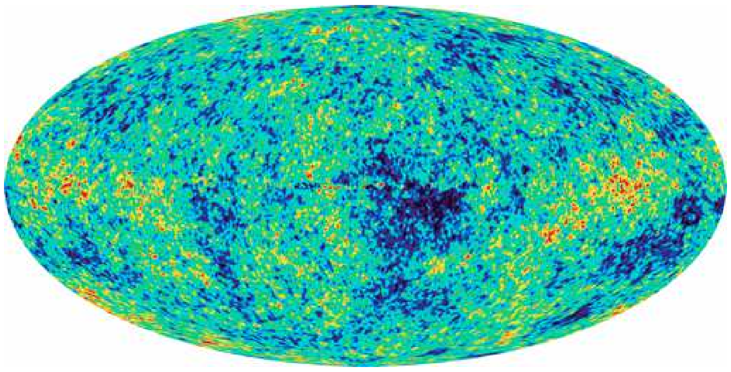


Figure 7.4: CMB temperature fluctuations given by WMAP on a 5-year observation over the full sky (2008). The average temperature is 2.725 Kelvin. Blue regions denote colder temperatures, whereas red regions denote warmer ones (these fluctuations are of the order 10^{-5}).

certainly not isotropic. *Large Scale Structures of galaxies* appear in the universe: voids, walls and filaments. Those latter are thread-like structures containing galaxies bound together by gravitation. However, there isn't enough baryonic matter (matter we can "see and touch in everyday life") in the universe to explain the motions of the galaxies. Hence the need for an invisible matter, *dark matter*, in the models, to compensate the lack of baryonic elements. Figure 7.5 presents the skeleton obtained in a simulation of dark matter.

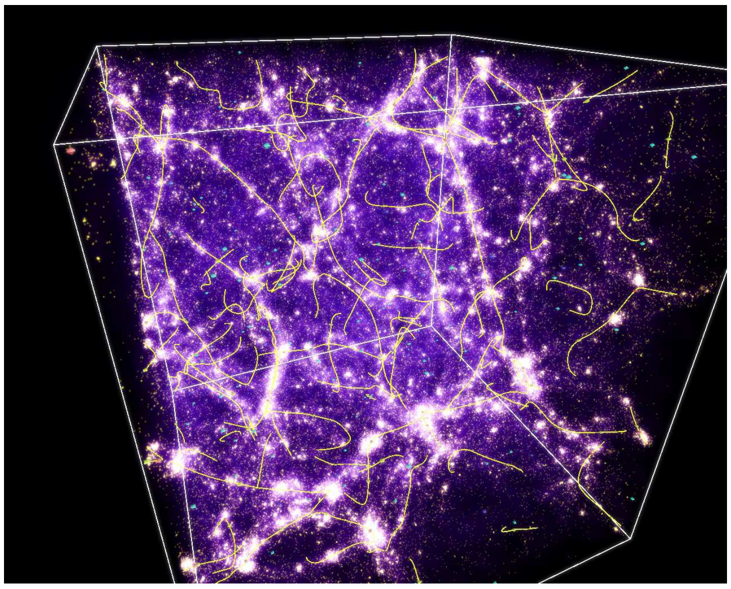


Figure 7.5: Universe skeleton as detected in a 50 Mpc dark matter simulation (© Thierry Sousbie, CRAL Lyon, France).

Velocities of galaxies are hard to determine, and are at the core of cosmological studies, as they allow to determine matter distribution in the universe. Indeed, the observed velocity of a galaxy ($V_{observed}$) is the composition of two vectors: the expansion velocity ($V_{expansion}$), and the gravitational pulls ($V_{gravitation}$):

$$V_{observed} = V_{expansion} + V_{gravitation}$$

However, an important point when determining velocities from a particular galaxy, is that only its radial component is attainable through observations. Indeed, as the distance measurements rely on the *Doppler Effect*, only the component being on the line of sight between the observer and the galaxy can be determined. Two observations made from two distant points would be necessary to be able to obtain the tangential component. However, during a one year rotation of the earth around the sun, the two most distant points are only 1UA apart, which is not sufficient. Hence, we need to rely on simulations to reconstruct the other non-radial components as well as the distribution of the dark matter which cannot be detected in the observations.

Determining the distance to a galaxy allows the physicists to get rid of the expansion velocity, as this latter is expressed as the universe expansion rate from the Hubble Constant H_0 multiplied by the distance d to the galaxy:

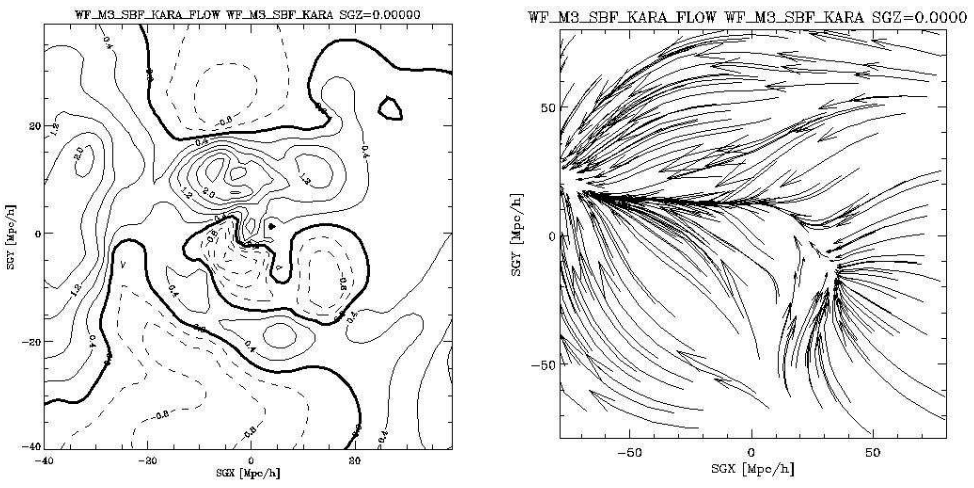
$$V_{expansion} = H_0 \times d$$

Two independent observations are needed to determine both the distance to a galaxy and its radial velocity observed. The distance allows to subtract the component due to the expansion of space, leaving the physicists with velocities due to the gravitational environment, called *peculiar velocities*, which directly probe the underlying gravitational field or matter distribution:

Peculiar velocities trace the full gravitation at each position of the universe, *i.e.,* dark and visible matter. Only the radial part of these peculiar velocities' vector are accessible from the observations. The Wiener filter permits to reconstruct the full 3D vector, and thus to build the full 3D velocity field and underlying 3D matter density field causing those velocities.

7.2.2 Initial Conditions Generation

Once the distances to every object is obtained, *i.e.,* the observational catalogs, we can derive the density field (*i.e.,* the number of galaxies present in a given direction), and the velocity field (velocity of each galaxy). However, having the velocity of each object is too complex and "noisy", thus data is first filtered. Figures 7.6a and 7.6b present an example of density and velocity fields used to generate IC.



(a) Density field (b) Velocity field

Figure 7.6: Density and velocity fields obtained once the Wiener filter applied.

At this point two possibilities exist. Either we generate initial conditions using only the CMB provided by WMAP5, and run the simulator on lots of generated IC in order to derive statistics on all the simulations. Or, we use the CMB, and the density and velocity fields in order to generate constrained IC that correspond to the observations. As time cannot be reverted, the method to generate IC from the fields is not the inverse method from running the simulation from an IC to the present day. In order to generate constrained IC, an analytical method is used, whereas the simulator uses a numerical approach when running the simulation from the IC to the present day. The parameters associated with an IC are the size of the box representing the universe in Megaparsecs (Mpc, one parsec is approximately 3.262 light-years), and the number of particles in the box.

7.2.3 Dark Matter Simulations

Cosmological simulations consist in evolving the particles present in the universe through cosmic time with regards to a particular model and parameterization. Whereas in the observations we only have access to the baryonic matter (which represents only 0.05% of the matter present in the universe, which in turn represents only about 0.3% of all the energy, the rest being dark energy), in simulations we can also take into account dark matter. Thus, we have access to much more information than what is really available through observations.

Due to the different scales that are involved in such simulations (scale of the universe versus the scale of the particles), an *Adaptive Mesh Refinement* technique is often used in classical cosmological simulators. This method increases the computation precision only on local parts of the universe, where particles interactions take place. Examples of such simulators are Gadget [151] or Ramses [160].

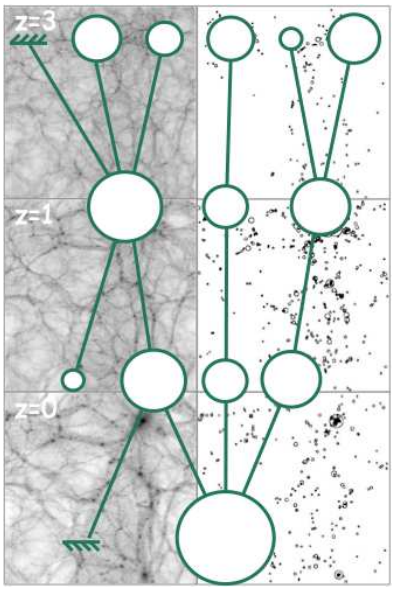
7.2.4 Mock Galaxies Catalogs

Once the simulations of dark matter are done, we need to create galaxies out of the dark matter distribution. GaLICS [106] (Galaxies In Cosmological Simulations), starting from theoretical priors, uses numerical simulations to describe hierarchical particles clustering. It uses a hybrid approach which first extracts halos (groups of particles) merging history trees from the dark matter simulations, and then uses a semi-analytical model to evolve galaxies in these trees. In fact, GaLICS is composed of three software used one after the other. HaloMaker detects the halos of dark matter using the 'friend-of-friend' (FOF) algorithm [73] (this algorithm links two particles in the same group if their distance is less than a certain linking length). TreeMaker builds the merger trees: the merging tree of the dark matter halos formed thanks to the accretion of matter, Figure 7.7a presents a merger tree. Finally, GalaxyMaker creates galaxies out of the merger trees.

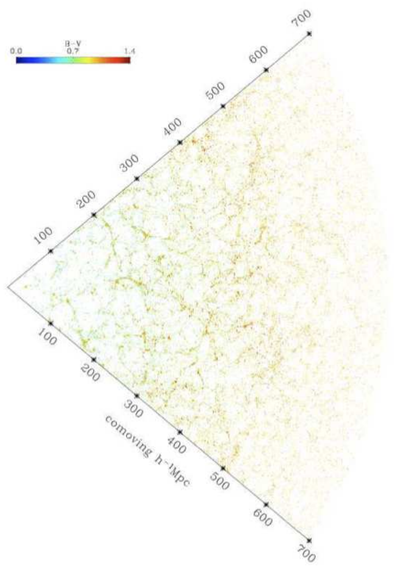
However, having the position and properties of the galaxies isn't sufficient to be able to compare the simulations to the real data. We need to have catalogs of virtual observations, *i.e.,* catalogs that would describe the galaxies in the same way real observations would be able to describe them. MoMaF [50] (Mock Map Facility) creates mock cones of observations from the output of GaLICS. Given a line of sight, and an opening angle, MoMaF creates mock catalogs of galaxies, and also two other kinds of mocks: *pre-observation maps* which are projection of the galaxies on the sky, and *post-observation maps* which include realistic modeling of the characteristics of the instruments used to observe the sky, hence it includes the bias one would observe with these instruments. Figure 7.7b presents a mock catalog.

7.2.5 Observations/Simulations Comparison

The last step consists in comparing the results obtained in the simulations to the real observations. There is currently no automated process for this step, and is thus still conducted manually. Either statistical studies are conducted, or precise studies on the structures of the universe in the case of constrained simulations. Ultimately, when a simulation is coherent with the observation, it allows to find all the missing parameters in the observations: tangential velocity, distribution of dark matter... A simulation is valid if the number of galaxies, their magnitude and spectrum match the observations. If not, then the physical processes modelizations in GaLICS are questioned, and modified for future simulations.



(a) A merger tree



(b) Mock 2dFGRS: each point represents a galaxy

Figure 7.7: Merger tree, and mock galaxy catalog.

7.3 Running Cosmological Simulations with Diet

In this part, we present our study of cosmological simulations. We first present, in Sections 7.3.1 and 7.3.2, modelization results on dark matter simulations and their post-processing. These performance estimations are used in our Diet client/server implementation in order to predict the execution time of the services. We then present the whole cosmological simulations workflow in Section 7.3.3, before giving experimental results of the first prototype in Section 7.3.4, and then results of a large scale experiment in Section 7.3.5.

7.3.1 Dark Matter Simulations

Ramses

Ramses is a typical computational intensive application used by astrophysicists to study the formation of galaxies. It is used, among other things, to simulate the evolution through cosmic time of a collisionless, self-gravitating fluid called *dark matter*. Individual trajectories of macro-particles are integrated using a state-of-the-art *N body solver*, coupled to a finite volume Euler solver, based on the Adaptive Mesh Refinement techniques. The computational space is decomposed among the available processors using a *mesh partitioning* strategy based on the Peano–Hilbert cell ordering. Ramses is a parallel program based on MPI [149] (Message Passing Interface), and has already been used on more than 2000 processors on the Mare Nostrum cluster on a very large scale simulation. The simulations we are tackling are much smaller, and do not have the same goal. While large scale simulations (on Mare Nostrum the IC contained 1024^3 particles) aim at having the finest precision, the simulations we run on the grid aim at obtaining statistical results from different IC with a lower resolution (less than 256^3 particles).

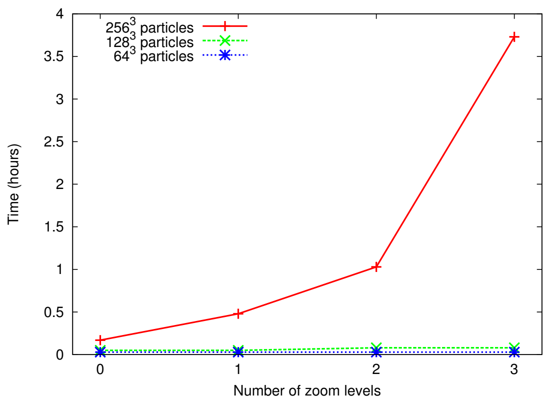
Simulations input/output

Ramses reads the initial conditions from Fortran binary files, which are generated using a modified version of the Grafic2 [46] code. This application generates Gaussian random fields at different resolution levels, consistent with current observational data obtained by the WMAP5 satellite observing the CMB radiation. The generated IC can be of two sorts. Either it contains a single level of resolution, *i.e.,* the universe has a “homogeneous”distribution of dark matter, these IC are used to perform the initial low resolution simulation of a zoom re-simulation. Or, it can generate multiple levels of resolution, *i.e.,* several nested “boxes”, like matryoshka dolls. These nested boxes add more particles, and thus more precision locally, on a point of interest in the universe. These are used in the zoom part of the simulation.

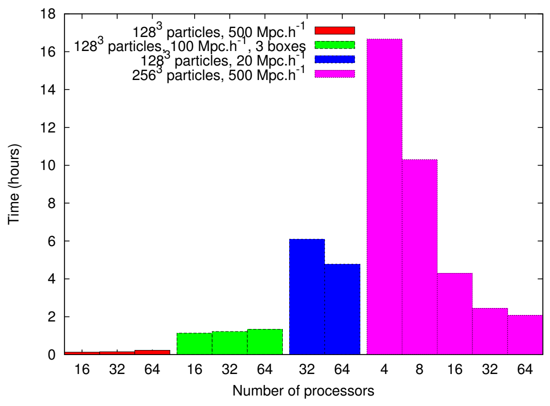
The outputs of Ramses are twofold. Periodically, Ramses outputs backup files so as to be able to restart a simulation at a given time step. The second kind of output is of course the result of the simulation. The user defines time epochs at which she would like to obtain a *snapshot* of the simulation, *i.e.,* a description of all particles (their position, mass, velocity. . .), Ramses will then output as many snapshots as requested.

Execution time and I/O size

Figures 7.8a and 7.8b respectively present the execution time for generating IC with Grafic2, and the dark matter simulation time with Ramses. Grafic2 execution time slightly depends on the number of particles in the IC when no nested boxes are generated. However, if adding several nested boxes into low resolution IC (with less than 256^3 particles) does not increase the execution time, their generation time becomes paramount for higher resolutions (higher than or equal to 256^3 particles). Ramses execution time does not benefit from the code parallelization on low resolution simulations. However, on higher resolution IC, the simulation time is “roughly” divided by two whenever the number of processors is doubled. The execution time also depends on the size of the universe, and is inversely proportional to its size in Mpc: the smaller, the denser the universe is, the longer the simulation will be, as more interactions will take place between particles.



(a) Grafic2: IC generation time



(b) Ramses: dark matter simulation time

Figure 7.8: Grafic2 and Ramses execution time.

Number of Particles	IC Size (GB)
2048^3	128
1024^3	16
512^3	2

Number of Particles	IC Size (MB)
256^3	256
128^3	32
64^3	4

Table 7.1: Initial conditions size. The given figures concern one level of resolution.

The input file size is easy to determine, as it grows linearly with the number of particles. The size of the universe does not impact the IC size. However, it has to be noted that these sizes concern each level of zoom. Thus, when running a zoom simulation on a box containing 256^3 particles and 3 levels of zoom, we have a total of 1GB (the level 0, and 3 more levels of zoom). Table 7.1 presents the IC sizes.

Concerning output files; a backup file is in fact a copy of the memory used by the program at a given time, so its size depends on which time step the simulation is, and the size of the input, but typically, a backup file weights a few gigabytes. As the snapshots contain the parameters of all particles, each snapshot has roughly the same size as the IC, and even though the size of the snapshots tends to grow a little as the simulation reaches the present time, the IC size stays a good estimate.

7.3.2 Post-Processing

Dark matter simulations snapshots need to be post-processed, as they only contain information on the dark matter density field. Obtaining mock catalogs of galaxies requires further processing. We first need to run GalaxyMaker to form galaxies out of dark matter, and then MoMaF to obtain mock catalogs. However, these software rely on "hand-made" galaxy formation models, whose results depend a lot on the parameterization. So, in order to fine-tune their models, the physicists need to explore the whole range of parameters combinations.

GaLICS and MoMaF

Actually, as explained in Section 7.2.4, three applications, which are part of the GaLICS suite, need to be executed on the snapshots in order to obtain galaxies. The first one, HaloMaker, can be executed in parallel on all the snapshots: it detects halos of dark matter, *i.e.,* density peaks. Then, TreeMaker gathers all the outputs of HaloMaker, and builds the merger tree, *i.e.,* the tree representing the history of halos formation. These two phases are quite straightforward, and need only to be run once. However, the results of the last step, GalaxyMaker, are highly dependent on the physical model parameterization, which needs to be explored so as to determine which sets of parameters are relevant. Thus, we only concentrate on this last part of the post-processing, which need in turn to be transformed into mock observational catalogs with MoMaF. Even though MoMaF results do not depend on the physical model parameterization, we need to produce many observational *cones* (simulations of what a real instrument would see) with many opening angles and lines of sight.

Post-processing execution time and output size

As post-processing applications are meant to be executed in a parameter sweep manner (so as to analyze the influence of each parameter on the galaxies' formation), we need to be able to provide the best possible estimation on the execution time and generated files' size, in order to be able to predict the whole post-processing execution time. This is especially useful in the server selection phase when a request is submitted to the middleware. It also helps the servers reserve the correct amount of resource time, when the server is in charge of, for example, a batch scheduler. Thus, we need to analyze the influence of each parameter on the applications' behavior.

Parameters influencing the galaxies formation results are the following: the star formation efficiency, α_{par}, the feedback efficiency, ϵ, the halo recycling efficiency, υ, and whether the supernova feedback model is used or not, S_f. Another parameter which is not in itself part of the model, but which influences the execution time and the output files' size, is the size of the input file which contains all the halos and their formation history.

We ran benchmarks on GalaxyMaker in order to derive models for execution time and output files' size. We used four files generated by TreeMaker as inputs for GalaxyMaker: a small one (3MB), and three larger ones (10, 38 and 120MB). We tested a representative set of values for each parameter, leading to 250 executions for each input files.

Unfortunately, we did not manage to obtain accurate execution time and output files' size models for MoMaF, as this application's behavior is much more erratic than GalaxyMaker's. The only relevant information we extracted from our benchmarks is that the computation time increases with the size of the observational cone (the larger it is, the more galaxies it contains), and its depth of view. However, no fine correlations could be found between the parameters' value and the execution time, or the output files' size. Thus, we now only present GalaxyMaker's execution time and output files' size models.

Execution time. Equation (7.1) presents the execution time model for GalaxyMaker.

$$T_{\text{GalaxyMaker}} = A_t \times nb_{halos} + B_t + c_t \times \left(\frac{\alpha_{par}}{\epsilon}\right)^{d_t} \tag{7.1}$$

Where $T_{\text{GalaxyMaker}}$ is GalaxyMaker execution time in seconds, nb_{halos} is the number of halos found in the input file (*i.e.,* this value is linearly proportional to the input file's size), A_t and B_t are constants, and c_t and d_t are linear functions of nb_{halos}, whose constants values depend on whether S_f is true or false. As can be seen, v does not appear in the model, as its influence on the execution time is negligible compared to the influence of the other parameters. Figure 7.9 presents the ratio between the execution time given by the model, and the real execution time. As can be seen, for small input files, the model overestimates the execution time, this is often the case when modeling application behavior on small inputs: there is less swapping and caching problems.

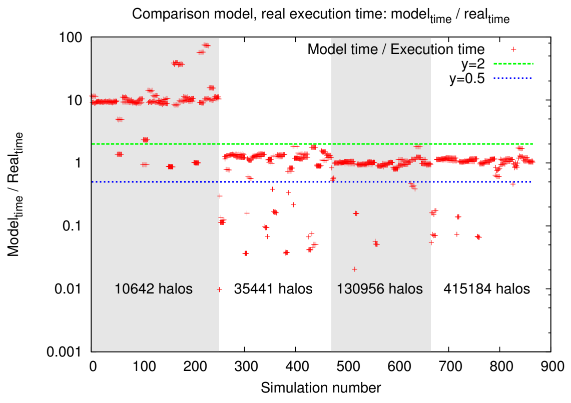


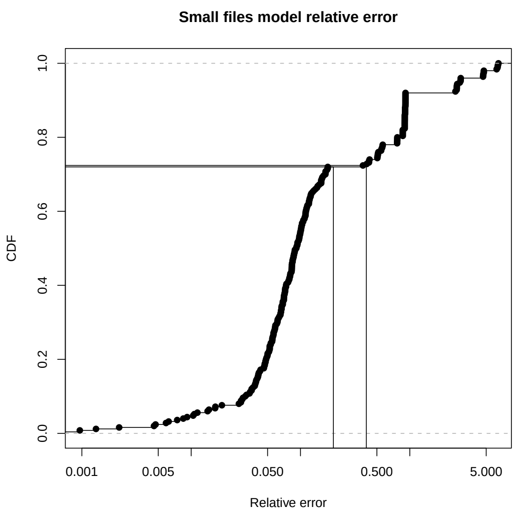
Figure 7.9: GalaxyMaker computation time model.

Concerning the smaller input file (10642 halos), the execution time is much lower than what the model predicts. This is due to cache mechanisms: as the data is much smaller, it best fits into the cache, which reduces the number of cache misses, and incidentally reduces the computation time. The model returns roughly a computation time ten times bigger than the real execution time. If we divide by 10 the result of the model, we obtain a relative error that is lower than 20% for 72% of the 250 experiments on the smallest file. As for the larger files, we have 58.5% of the 615 experiments which have a relative error smaller than 20%, and 81.5% a relative error smaller than 40%. Figures 7.10a and 7.10b respectively present the cumulative distribution function (CDF) of the relative error for the small input file when dividing the model execution time value by 10, and for the large files.

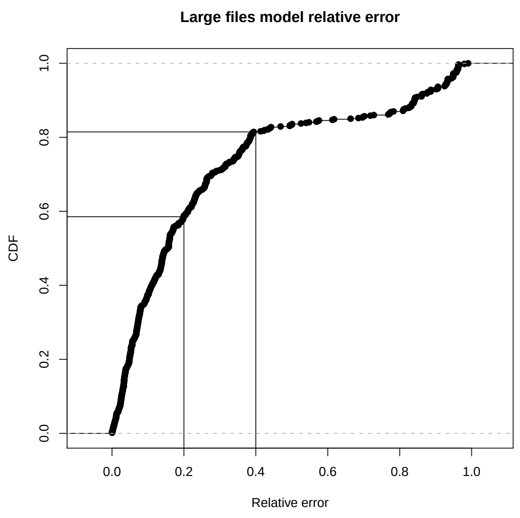
We tested our model against two other input files, a large one (95995 halos), and a small one (12558 halos). With the large file, the model predicts the execution time within 20% of relative error for 73.0% of the experiments (196 experiments), and 90.3% within 40% of relative error. For the smaller file, dividing the model result by 10 provides a prediction within 20% of relative error for 69.5% of the experiments (240 experiments), and 79.2% within 40% of relative error. The varying number of experiments is due to the fact that for some parameters' combinations, GalaxyMaker was not able to create galaxies, and crashed, thus we removed these failed experiments. CDF for the small and large test files can be respectively seen in Figures 7.11a and 7.11b.

Outputs. The output files' size is almost constant for a given input file, and thus easier to model: it only depends on the input file's size, *i.e.,* the number of halos. Equation (7.2) presents the model for output file size, and Figure 7.12a the comparison between the model and the real output files' size.

$$S_{\text{GalaxyMaker}} = A_s \times nb_{halos} + B_s \tag{7.2}$$

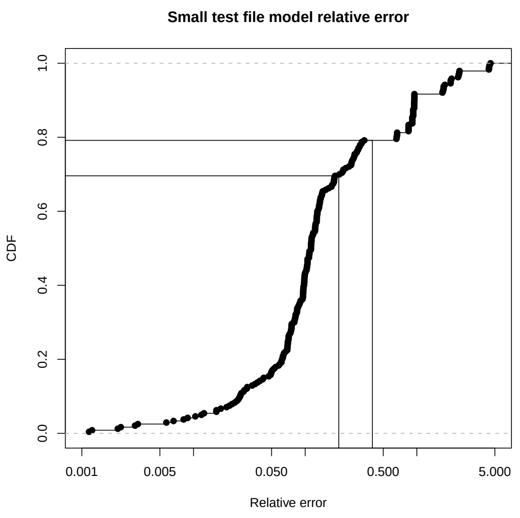


(a) Small file CDF

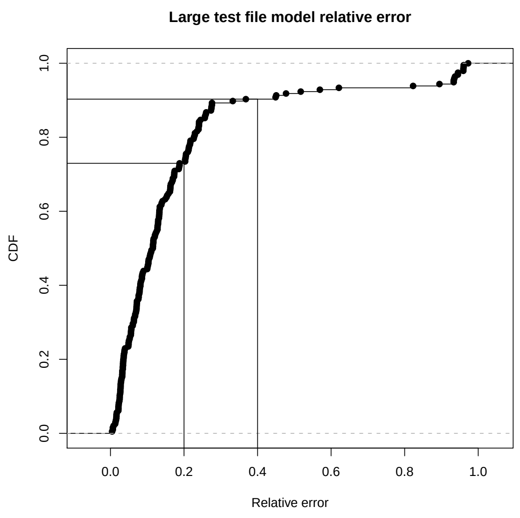


(b) Large files CDF

Figure 7.10: GalaxyMaker execution time model, relative error CDF.



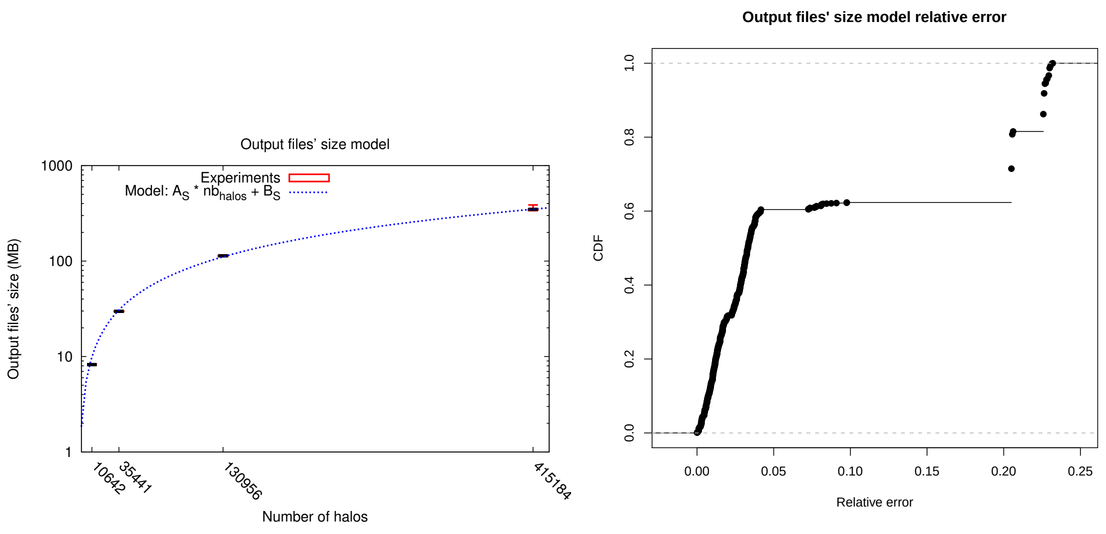
(a) Small test file CDF



(b) Large test files CDF

Figure 7.11: GalaxyMaker execution time model verification, relative error CDF.

Where $S_{\text{GalaxyMaker}}$ is the output files' size in MB, A_s and B_s are constants. The output files' size is really stable, and thus the model closely matches the real output size: apart from the smallest input files for which the model gives a relative error around 20% $\sim$ 23.2%, the relative errors are lower than 4% for about 60% of the 1301 tested cases. Figure 7.12b presents the CDF of the output files' size model relative error, for both the four benchmarked input tree files, and the two test tree files.



(a) Model (b) Relative error CDF

Figure 7.12: GalaxyMaker output files' size model.

7.3.3 Simulation Workflow

In this section, we present the whole workflow for cosmological simulations. The workflow is depicted in Figure 7.13. It is divided into two main parts: the dark matter simulation (*i.e.,* IC generation with Grafic2, and the simulation itself with Ramses) and the "one-shot" post-process (*i.e.,* HaloMaker and TreeMaker), followed by the parameter sweep part with GalaxyMaker and MoMaF. Three Diet services are dedicated to executing this workflow. The first one deals with the dark matter simulation itself along with HaloMaker and TreeMaker post-processing, and can possibly run GalaxyMaker, if no parameter sweep is requested. The second service executes an instance of GalaxyMaker, thus, during the parameter sweep part, we have $x \times |\{parameters\}|$ calls to the GalaxyMaker service (x being the number of files that TreeMaker created, and $\{parameters\}$ the set of tested parameters). The same number of calls holds for the last service, which is in charge of running MoMaF. This workflow is submitted by the Diet client to the MA_{DAG}, which in turn manages the execution of the different services.

As the first part of the simulation uses a parallel program, Ramses, the first service is already in charge of several processors. Thus, in order to reduce the communications, the IC generation, the dark matter simulation and the execution of HaloMaker and TreeMaker are grouped into one single service. The parallel calls to HaloMaker can easily be done by re-using the same processors dedicated to Ramses.

In fact, more than three services are requested to run cosmological simulations. As each step creates a lot of temporary data, and in order to reduce the communications, all files used and created by a given service have to be kept on the platform, for as long as another service might need them. Thus, whenever a service ends, it cannot necessarily delete the files it created. Moreover, the way Diet works prevents the client from knowing that a file is no longer used by any services: the MA_{DAG} takes the decision of which task has to be executed, then sends an order to the client telling it to request this selected service. Thus, the client only knows about the currently executed services, but nothing about the whole workflow progress. Besides, the client does not know the name of these temporary files, nor their location on the platform (which machines hold them). The solution adopted to deal with this problem, and remove all temporary files once they are not needed anymore, is to spawn a new *cleanup service* whenever one of the three main services finishes, and return the name of the service to the client. These new cleanup services do not require any parameter, and have a unique identifier, which is a Universally Unique Identifier

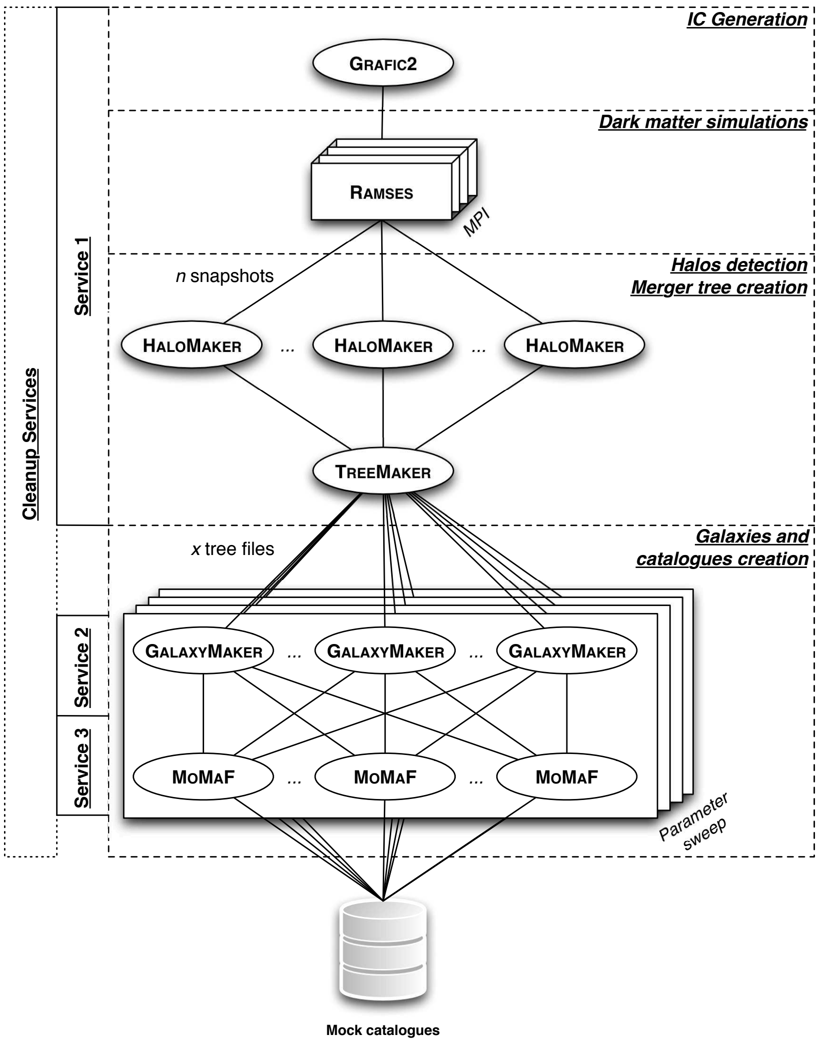


Figure 7.13: Cosmological simulations workflow.

(UUID). Thus, a cleanup service uniquely represents the files created on a specific machine, by a specific service. In fact, when we create the workflow description XML, we already add those services with the correct dependencies. The point that we do not know beforehand the name of the services is not a problem, as we added into DIET the capability to use the output variables of a DIET service, as input for the service name of one of the children task in the workflow. Thus, whenever a service is created, its correct name is replaced in the workflow description, which enables the MA_{DAG} to execute this new service. When a cleanup service is called, it removes all temporary data it is in charge of, and finally deletes itself so that it cannot be called anymore. Originally, DIET does not handle such dynamic service spawning and deletion, only services declared at the SED initialization were possible. We added the possibility to dynamically add and remove services. The new services can either be described by an already existing service, or they can also be loaded from an external library.

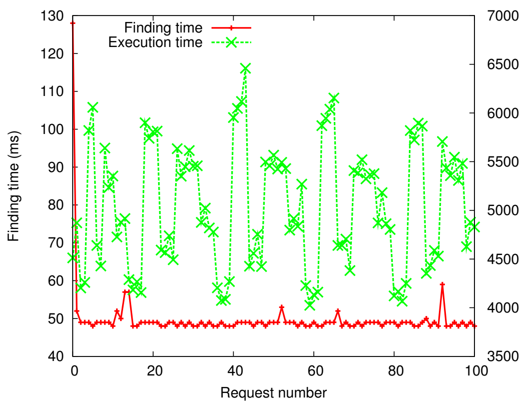
Finally, the MA_{DAG} relies on the Heterogeneous Earliest Finish Time [161] (HEFT) heuristic to schedule the different tasks. As HEFT selects the server that can execute the task with the minimum finish time, we need to be able to predict the finish time of a task on a given server. The finish time of a task is the sum of the waiting time (time to finish all previously enqueued task on the server) and the execution time. Thus, we used the previously presented models (or a rough estimate when no model was available) to give an estimate of the execution time of each task. In order to cope with the models' uncertainties, we compute a sliding mean on the relative error whenever a job is executed. This mean is then used to compensate the errors of the model.

Actually, the whole workflow is seldom executed. It is often divided into two parts: the dark matter simulation, and the post-processing phase. The physicists often first simulate the execution of dark matter, then run a few analysis, before effectively running the post-processing. Thus, there usually is a gap between the two parts. Moreover, even the parameter sweep part is more or less run interactively: a few sets of parameters are tested, and depending on the results, new sets are generated and tested. Thus, the parameters values' range is studied iteratively.

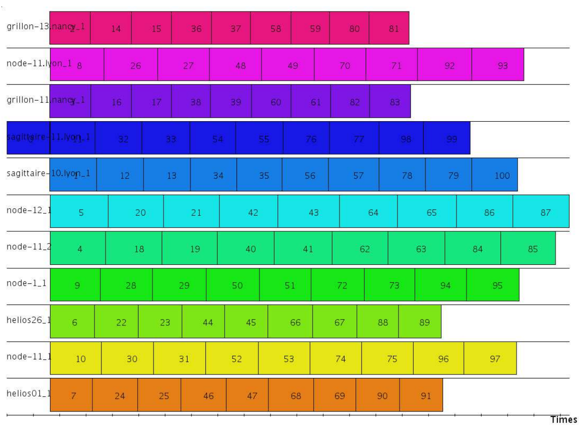
7.3.4 Prototype

The first implemented prototype aimed at ascertaining the feasibility of porting GRAFIC2, RAMSES and GALICS on a grid environment: Grid'5000, with the help of a grid middleware: DIET. To this end, we implemented a DIET client and server. The SED contained a single service which was in charge of a simplified workflow: MOMAF was not used, and only one instance of GALAXYMAKER was executed. The idea was to simulate the execution of a real zoom simulation on a low resolution universe (128^3 particles, in a $100 Mpc.h^{-1}$ box). Hence, the "global workflow" consisted in running one dark matter simulation, and then, running one hundred sub-simulations with nested boxes IC so as to have a better resolution locally on some points of the universe.

DIET was deployed, along with the cosmological applications, on five sites of Grid'5000. The hierarchy comprised a Master Agent in Lyon, six Local Agents (two in Lyon, and one in Lille, Nancy, Toulouse and Sophia), and eleven SEDs (two per site, apart from Lyon, which had three). Each SED was in charge of 16 machines in order to be able to run RAMSES using MPI.



(a) Finding and execution times



(b) Gantt chart

Figure 7.14: Prototype: SED finding time, requests execution time, and Gantt chart.

Figure 7.14a presents the requests' *finding time*, *i.e.,* the delay between the moment the client sent a request to the MA, and the time it received a reply with a suitable SED to execute the request, and the execution time of the simulation, for each simulation. The execution time variations observed between the different simulations (between 4000s and 6500s) come from the processors' speed differences. The finding time is really low compared to the simulation execution time, and almost constant: on average it took 49.8ms to find a suitable SED. Apart from the finding time, we need to add the overhead added by the service initialization: on average it took 20.8ms. Thus, the average overhead for one simulation is about 70.6ms, so a total of about 7s of overhead on a 16hrs run for 101 simulations. This result comforts us in our idea of using DIET as a good solution for running cosmological simulations on a grid.

The total execution time for the whole workflow was 16h 18min 43s (1h 15min 11s for the first part and an average of 1h 24min 1s for each simulation in the second part). Figure 7.14b presents the Gantt chart of the whole workflow execution: each line represents a SED, and each "colored box" represents the execution time of a simulation. From this chart, we can see that the schedule is far from being optimum. Indeed, the fastest SeDs finish computing a few hours before the slowest one. The reason for this is twofold. Firstly, after the first simulation is finished, we have 100 sub-simulations to schedule all at once. As we used the default Diet scheduler, which amounts to allocating the requests in a Round-Robin fashion, each SeD receives an equal share of the total workload in terms of number of requests to execute. Secondly, as we did not use the workflow system for this prototype (it was not fully implemented at this time), the jobs are not being held by the system before being scheduled, *i.e.,* once the client can submit the 100 tasks, it submits them without waiting for any resource to be free. These two conditions lead to a poor schedule. However, using distributed resources helped computing the simulations faster, would they have been executed sequentially, it would have taken more than 141hrs.

Once small scale experiments have been validated on a small platform, we needed to stress our prototype with longer runs, *i.e.,* simulations on higher resolution IC, and on a larger platform. This is the purpose of the next section.

7.3.5 Large Scale Experiment

In order to validate our prototype to run cosmological simulations, we realized a large scale experiment on Grid'5000[3]. The goal was to stress all the components of a simulation: the hardware, the middleware, and the simulation software. Thus, we reserved 979 machines, on 12 clusters on 7 sites of Grid'5000, for 48 hours. This represented the totality of Grid'5000 resources at the time, minus the resources that did not work properly (they were either dead or problems occurred on them in the first steps of the experiment). The simulation workflow was reduced so as to be composed of Grafic2 for the 256^3 particles, $100 Mpc.h^{-1}$ IC generation, followed by a Ramses simulation, and finally using a simple post-processing to obtain "slice pictures" of the universe on all snapshots. The approach was somewhat different from the previous experiment, as the goal here was to produce statistical data for the physicists: they did not want to study the formation of galaxies, but instead the formation of large scale structures, *i.e.,* the universe's skeleton. So, independent simulations were run with exactly the same cosmological parameters. The only difference came from the parameters fed into Grafic2 to generate the IC.

Figure 7.15 presents the deployed Diet hierarchy. It was composed of one MA, 12 LA (one per site), and 29 SeDs. Each SeD was limited to one execution of Ramses at a time. On the whole, it was 816 nodes that were dedicated to the simulations (some nodes among the 979 did not work properly). A few snapshots of the universe produced by one of the simulations can be seen on Figure 7.16.

The jobs submission mechanism was different from the previous experiment. All simulations were independent, and they were not released all at once by the client. Instead, the client was allowed to submit a new simulation request, if and only if a SeD was free. Thus, at any time, there were at most 29 simulations running concurrently on the platform.

The results were concluding, as we managed to submit 59 simulations (33 were completed, and 26 were partially made), corresponding to 193 GB of data, and we obtained an equivalent of 368 days of computation time on one processor, and this realized in 28 hours of computation. Compared to the platform the physicists usually use for their experiment at the CEA, it would have taken more than 6 months of computation. This work allowed to improve the stability of Grid'5000, to improve data management within Diet, and produced useful simulation data for the cosmologists.

3. This work received the "Best Large Scale Experiment" award during the 2009 Grid'5000 Spring School.

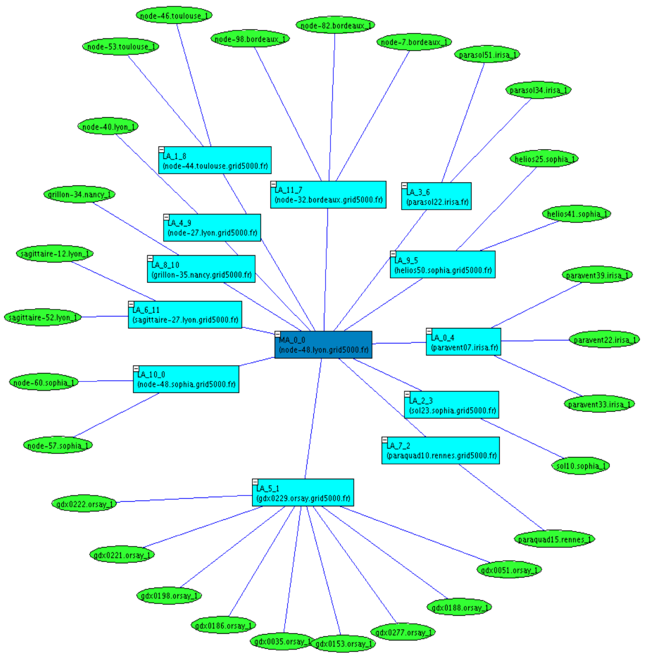


Figure 7.15: Large scale experiment: DIET hierarchy.

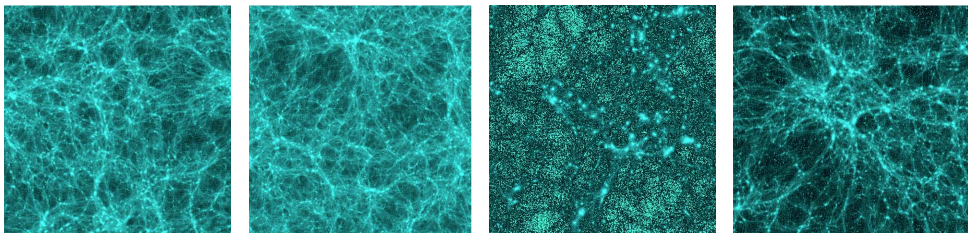


Figure 7.16: Dark matter density taken at four stages of the universe evolution.

7.4 Conclusion

In this chapter, we have first presented our end-to-end infrastructure to run scientific applications in a transparent manner on a grid platform. Our solution relies on a web site on the user side to submit jobs. Those latter are fed into DIET, which takes care of creating the relevant workflow, and manages the tasks and the data. Finally, in order to abstract the complexity of the network, and especially of the firewalls, we rely on PadicoTM. Then, we presented background on cosmological studies based on simulations, these simulations are essential in the process of analyzing the distant galaxies, as they allow to determine parameters that are not attainable through observations. The last section provides an analysis on the software used for cosmological simulations: their execution time and the size of the generated files. We also provide an efficient framework to run and analyze simulations, based on DIET and its workflow support: DIET can handle the whole workflow execution, and manages temporary data thanks to dynamic services. Our experiments on a large scale platform gave encouraging results.

Conclusion and Future Works

Conclusion

Grids play an important role for scientific computing. While they aim at gathering heterogeneous and distributed resources to provide a large amount of computational power, their utilization is still far from being simple. Deploying services on a grid, accessing them efficiently, and in a transparent manner is a complex process. Several barriers are still hindering this process. These problems are inherent to this kind of platforms, mainly related to their heterogeneity, scale, and dynamic nature.

In this dissertation, we have presented what we think are some of the end-to-end requirements to provide efficient, and transparent access to services deployed on a grid. The proposed solutions deal with the users' side of the problem, as well as with the administrators' side.

The purpose of the first chapter was to ease the understanding of the whole grid computing context, and to provide background on Grid-RPC middleware, and their deployment on such heterogeneous and distributed platforms. The first tackled problem, presented in Chapter 2, was the deployment of hierarchical network enabled server environments on a grid. We first gave a generic model for understanding the computation and communication costs present in a hierarchical middleware, when several services need to be deployed. Based on this model, we proposed three heuristics built on linear programs, and genetic algorithm techniques, for three different platforms, namely fully homogeneous, communication homogeneous/computation heterogeneous, and fully heterogeneous platforms. We showed the effectiveness of our approach through real experiments on the French research grid, Grid'5000, with the Diet middleware, by comparing our solutions with classical deployment approaches.

Grids being heterogeneous, dynamic and error prone environments, we then moved on to the problem of grouping nodes using a fault-tolerant approach. In Chapter 3, we have presented related work on clustering techniques, as well as an introduction to the concept of self-stabilization. Then, in Chapter 4, we focused on the problem of grouping nodes into clusters of bounded diameter, so as to guaranty communications' response time within each cluster. To the best of our knowledge, we were the first to address this problem on weighted graphs, which are a more pertinent representation of a grid network topology than unweighted graphs, which are relevant for MANET. Our solution, K-CLUSTERING, is totally distributed, and self-stabilizing under an unfair daemon. K-CLUSTERING is built upon three self-stabilizing algorithms, and is, as provided by the self-stabilization paradigm, able to provide a correct clustering after crashes, wrong initializations, or any transient faults affecting the system. It has very low memory requirements, at the expense of a poor theoretical convergence time. However, simulation results have shown that in practice, the convergence time can be several order of magnitude lower than the theoretical bound. We also gave the conditions under which a composition of self-stabilizing algorithms forms a self-stabilizing algorithm under the unfair daemon.

Once the problem of accessing services efficiently has been tackled, remained the problem of scheduling the different jobs. Models, objectives and techniques related to scheduling have been discussed in Chapter 5. The problem of scheduling bags-of-tasks on a platform composed of several clusters, with constraints on the computational power the users are allowed to use, have been presented in Chapter 6. We have studied two cases: when tasks are all released at the same time, and when release dates are taken into account. Thus, two objectives have been considered: maximizing the number of tasks scheduled per application, and the maximum stretch that the tasks can undergo. Our solutions are based on linear programs. We iteratively built the solutions, and showed at each step the limitations of each formulation. We have presented extensions to the linear programs to circumvent the encountered problems.

Finally, in Chapter 7, we discussed more practical contributions, applied to cosmological simulations. We have first presented a complete infrastructure to transparently run scientific applications on a grid. It relies on three layers. At the communication level, PadicoTM takes care of routing and tunneling whenever required to interconnect several computing elements. At an intermediary level, the DIET middleware is used to manage workflows' executions, and data management. Finally, a web-site serves as an front end between the users' needs and DIET. The whole complexity of using the grid is thus hidden to the end-users. Then, we specialize this infrastructure for cosmological simulations. The behavior of the cosmological applications have been studied, and the corresponding workflows can be executed using DIET. Experiments on Grid'5000 show the feasibility and effectiveness of our approach.

Thus, submitting jobs to a grid can efficiently be managed from the web-site, down to the servers. A request goes through a middleware's hierarchy, which shape is tuned especially for the platform, and the applications; the deployment taking place on clusters with bounded communication time. Finally, the tasks can be scheduled on clusters, while respecting constraints regarding the available computing power.

Future Works

Several improvements can be undertaken on the solutions presented in this thesis.

– The deployment algorithms presented in Part I are only able to deal with fully connected platforms. A deeper study on the network topology is here required to take into account contentions, and more accurate and realistic communication costs.
– Still in Part I, another important point, is that our algorithms can only deal with a *static* deployment, *i.e.,* we know in advance all the parameters of the problem, and provide a hierarchy that best fits the requirements. What if the users' needs change over time, or if nodes' availability can change? We need *dynamic* algorithms to adapt the middleware to the evolution of the environment. We already added the possibility to dynamically modify the DIET hierarchy, but we still lack intelligent and automatic mechanisms. Recent works [105, 147] based on autonomic computing already propose a few repair, and adaptation mechanisms for DIET using TUNe. However, the current policies are quite simple. In the context of errors and crashes, another interesting work would be to make a self-stabilizing version of the dynamic deployment algorithms, and implement them in TUNe for example.
– The design of our self-stabilizing k-clustering algorithm presented in Part II is highly dependent on the ID of the processes. Thus, the quality of the result can greatly vary, depending on the IDs distribution. It could be interesting to modify the way the clusterheads are elected, to be less dependent on the IDs. Another important point that needs to be addressed is the complexity of the algorithm. Currently, it requires $O(nk)$ rounds to converge to a legitimate solution. This value can be quite high, and would be a drawback for the evolution of K-CLUSTERING into a message passing model. Indeed, the number of transmitted messages could become prohibitive. Reducing the complexity would certainly require to increase the memory requirements. Finally, the next step is also to couple this clustering algorithm with the deployment algorithms so as to provide guaranties on the response time of the middleware.
– In Part III, we present the scheduling of applications under a maximum utilization constraint. This problem tend in fact to schedule the tasks on the less constrained clusters. The next step is to provide online scheduling algorithms, and implement them into DIET plug-in schedulers. A feedback mechanism operating apart from the classical Grid-RPC messages would be required to help the dynamic adaptation of DIET. Based on scheduling results, the DIET hierarchy could be modified, or, in the case where the maximum utilization constraints are handled by DIET, they could be dynamically adapted depending on the obtained maximum stretch for example, or they could be coupled with energy aware scheduling algorithms.
– Concerning cosmological simulations presented in Part IV, specific schedulers could be developed

and plugged into Diet. Currently HEFT is used, but schedulers dedicated to cosmological simulations, based on the software's models could greatly improve the execution time. Moreover, only the execution time is taken into account, the communications are left aside. Due to the amount of data to transfer, it could be interesting to couple Diet with a performance prediction system to give an estimate of the transfer times. This could be done using a grid monitoring system (such as Pilgrim [19]) to retrieve values on the machines' load, and the network's capacity, coupled with a simulation tool (such as SimGrid [64]) to simulate executions and transfers. This would allow to simulate the contentions that are likely to occur when transferring large amounts of data on a distributed platform.

– Finally, in a more global vision, the last step would be to provide a software that would integrate everything into a single deployment and autonomic management tool. This tool would be an offspring of GoDiet and ADAGE for their platform description and deployment capabilities, and TUNe for its autonomic capabilities. The ideal would be to be able to provide a tool that would, given the platform and applications descriptions, provide an efficient hierarchy for the first deployment (based on the proposed algorithms, and the self-stabilizing clustering). This first deployment could require the *co-deployment* of PadicoTM in order to cope with networking problems. Then, using monitoring, and feedback from the schedulers, dynamically adapt the hierarchy. From the point of view of the administrator of the middleware, such a tool would be highly profitable.

In this thesis, we focused essentially on a particular type of platforms that are grids. Though grids are already quite complex environments, the next generation of platform is already there, and add *virtualization* on top of it all. *Cloud computing* [96] is the next step. And despite the fact that clouds aim at hiding the complexity of grids, they bring an even greater dynamicity and heterogeneity to the platforms.

The next step is thus to manage *elastic cloud computing* where virtual machines can come and go on demand, and also be migrated. Moreover, machines may no longer be dedicated to the middleware and its computations. Indeed, as clouds rely on virtualization, several virtual systems may run concurrently on a same physical machine. Diet is inclined to manage *SaaS* (Software as a Service) [61] platforms, by providing on demand resources. Managing SaaS platforms can be seen as dynamic deployment and adaptation of the middleware. The needs are the same than the ones addressed in this thesis: a model describing the behavior of the platform and the middleware, and efficient algorithms that take into account the various modifications of the environment.

List of Figures

References

[1] Diet Webboard, 2010. http://graal.ens-lyon.fr/DIET/dietwebboard.html.

[2] Directed Acyclic Graph Manager, 2010. http://www.cs.wisc.edu/condor/dagman.

[3] The Distributed ASCI Supercomputer 2 (DAS-2), 2010. http://www.cs.vu.nl/das2.

[4] The Distributed ASCI Supercomputer 3 (DAS-3), 2010. http://www.cs.vu.nl/das3/index.shtml.

[5] The EGEE Project, 2010. http://www.eu-egee.org.

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

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

[8] GNU Linear Programming Toolkit, 2010. http://www.gnu.org/software/glpk.

[9] The Grid Workloads Archive, 2010. http://gwa.ewi.tudelft.nl.

[10] The Horizon Project, 2010. http://www.projet-horizon.fr.

[11] ILOG CPLEX, 2010. http://www-01.ibm.com/software/integration/optimization/cplex.

[12] Kadeploy, 2010. http://kadeploy.imag.fr.

[13] LHC - The Large Hadron Collider, 2010. http://lhc.web.cern.ch/lhc.

[14] lplsolve 5.5, 2010. http://lpsolve.sourceforge.net/5.5.

[15] MeTiS: Family of Multilevel Partitioning Algorithms, 2010. http://glaros.dtc.umn.edu/gkhome/views/metis.

[16] The NAREGI Grid, 2010. http://www.naregi.org.

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

[18] Open Grid Forum, 2010. http://www.ogf.org.

[19] Pilgrim, 2010. https://gforge.inria.fr/projects/grimap.

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

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

[22] The Shared Hierarchical Academic Research Computing Network (SHARCNET), 2010. https://www.sharcnet.ca.

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

[24] WLCG - Worldwide LHC Computing Grid, 2010. http://lcg.web.cern.ch/lcg.

[25] The World Community Grid Project, 2010. http://www.worldcommunitygrid.org.

[26] Zoltan: Parallel Partitioning, Load Balancing and Data-Management Services, 2010. http://www.cs.sandia.gov/Zoltan.

[27] I. Abdul-Fatah and S. Majumdar. Performance of CORBA-Based Client-Server Architectures. *IEEE Transactions on Parallel and Distributed Systems*, 13:111–127, 2002.

[28] E. Alba, F. Almeida, M. Blesa, J. Cabeza, C. Cotta, M. Díaz, I. Dorta, J. Gabarró, C. León, J. Luna, L. Moreno, C. Pablos, J. Petit, A. Rojas, and F. Xhafa. MALLBA: A Library of Skeletons for Combinatorial Optimisation. *Euro-Par 2002 Parallel Processing*, pages 63–73, 2002.

[29] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic Local Alignment Search Tool. *Journal of Molecular Biology*, 215(3):403 – 410, 1990.

[30] P. Amestoy, I. Duff, and J.-Y. L'Excellent. MUMPS MUltifrontal Massively Parallel Solver Version 2.0, 1998.

[31] K. Amin, G. von Laszewski, M. Hategan, N. Zaluzec, S. Hampton, and A. Rossi. GridAnt: A Client-Controllable Grid Workflow System. *HICSS'04: Proceedings of the 37th Annual Hawaii International Conference on System Sciences - Track 7*, 07:70210c, 2004.

[32] A. D. Amis, R. Prakash, T. H. Vuong, and D. T. Huynh. Max-Min D-Cluster Formation in Wireless Ad Hoc Networks. In *INFOCOM 2000: 19th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, pages 32–41. 2000.

[33] D. P. Anderson. BOINC: A System for Public-Resource Computing and Storage. In *GRID'04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10. IEEE Computer Society, Washington, DC, USA, 2004. ISBN 0-7695-2256-4.

[34] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: an Experiment in Public-Resource Computing. *Communications of the ACM*, 45(11):56–61, 2002.

[35] E. Anderson, Z. Bai, and C. Bischof. *LAPACK Users' guide.* Society for Industrial Mathematics, 1999.

[36] M. Arenas, P. Collet, A. Eiben, M. Jelasity, J. Merelo, B. Paechter, M. Preuß, and M. Schoenauer. A Framework for Distributed Evolutionary Algorithms. *Parallel Problem Solving from Nature —PPSN VII*, pages 665–675, 2002.

[37] S. Banerjee and S. Khuller. A Clustering Scheme for Hierarchical Control in Multi-Hop Wireless Networks. *INFOCOM 2001: 20th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, 2:1028–1037 vol.2, 2001.

[38] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *SOSP'03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177. ACM, New York, NY, USA, 2003. ISBN 1-58113-757-5.

[39] S. Basagni. Distributed and Mobility-Adaptive Clustering for Multimedia Support in Multi-Hop Wireless Networks. In *Vehicular Technology Conference, 1999. VTC 1999 - Fall. IEEE VTS 50th*, volume 2, pages 889–893 vol.2. 1999.

[40] S. Basagni. Distributed Clustering for Ad Hoc Networks. In *ISPAN'99: Fourth International Symposium on Parallel Architectures, Algorithms, and Networks, 1999*, pages 310–315. 1999.

[41] O. Beaumont, H. Casanova, A. Legrand, Y. Robert, and Y. Yang. Scheduling Divisible Loads on Star and Tree Networks: Results and Open Problems. *IEEE Transactions on Parallel and Distributed Systems*, 16(3):207–218, 2005.

[42] D. Bein, A. K. Datta, C. R. Jagganagari, and V. Villain. A Self-stabilizing Link-Cluster Algorithm in Mobile Ad Hoc Networks. In *ISPAN '05: Proceedings of the 8th International Symposium on Parallel Architectures, Algorithms and Networks*, pages 436–441. IEEE Computer Society, Washington, DC, USA, 2005. ISBN 0-7695-2509-1.

[43] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *ATEC'05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41. USENIX Association, Berkeley, CA, USA, 2005.

[44] M. A. Bender, S. Chakrabarti, and S. Muthukrishnan. Flow and Stretch Metrics for Scheduling Continuous Job Streams. In *SODA'98: Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, pages 270–279. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1998. ISBN 0-89871-410-9.

[45] F. Berman, G. Fox, and A. J. G. Hey. *Grid Computing: Making the Global Infrastructure a Reality.* John Wiley & Sons, Inc., New York, NY, USA, 2003. ISBN 0470853190.

[46] E. Bertschinger. Grafic2: Multiscale Gaussian Random Fields for Cosmological Simulations, 2010. http://web.mit.edu/edbert.

[47] D. Bertsimas and D. Gamarnik. Asymptotically Optimal Algorithms for Job Shop Scheduling and Packet Routing. *Journal of Algorithms*, 33(2):296–318, 1999.

[48] V. Bharadwaj, T. G. Robertazzi, and D. Ghose. *Scheduling Divisible Loads in Parallel and Distributed Systems*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996. ISBN 0818675217.

[49] L. Blackford, A. Cleary, J. Choi, E. d'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, *et al. ScaLAPACK Users' Guide*. Society for Industrial Mathematics, 1997.

[50] J. Blaizot, Y. Wadadekar, B. Guiderdoni, S. Colombi, E. Bertin, F. R. Bouchet, J. E. G. Devriendt, and S. Hatton. MoMaF: The Mock Map Facility. *Monthly Notices of the Royal Astronomical Society*, 360:159–175, 2005.

[51] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi, and T. Irena. Grid'5000: a Large Scale and Highly Reconfigurable Experimental Grid Testbed. *International Journal of High Performance Computing Applications*, 20(4):481–494, 2006.

[52] R. Bolze, E. Caron, F. Desprez, G. Hoesch, and C. Pontvieux. A Monitoring and Visualization Tool and Its Application for a Network Enabled Server Platform. *Computational Science and Its Applications - ICCSA 2006*, pages 202–213, 2006.

[53] T. Brady. *SmartGridRPC: A New RPC Model for High Performance Grid Computing and its Implementation in SmartGridSolve*. Ph.D. thesis, University College Dublin, 2009.

[54] L. Broto, D. Hagimont, P. Stolf, N. Depalma, and S. Temate. Autonomic Management Policy Specification in TUNe. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pages 1658–1663. ACM, New York, NY, USA, 2008. ISBN 978-1-59593-753-7.

[55] A. Buble, L. Bulej, and P. Tuma. CORBA Benchmarking: A Course with Hidden Obstacles. In *IPDPS'03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 279.1. IEEE Computer Society, Washington, DC, USA, 2003. ISBN 0-7695-1926-1.

[56] S. Cahon, N. Melab, and E. G. Talbi. ParadisEO: A Framework for the Reusable Design of Parallel and Distributed Metaheuristics. *Journal of Heuristics*, 10(3):357–380, 05 2004.

[57] Y. Caniou, E. Caron, F. Desprez, H. Nakada, K. Seymour, and Y. Tanaka. High Performance GridRPC Middleware. In G. Gravvanis, J. Morrison, H. Arabnia, and D. Power, editors, *Grid Technology and Applications: Recent Developments*. Nova Science Publishers, 2009. At Prepress. Pub. Date: 2009, 2nd quarter. ISBN 978-1-60692-768-7.

[58] E. Caron, P. K. Chouhan, and H. Dail. GoDiet: A Deployment Tool for Distributed Middleware on Grid'5000. In IEEE, editor, *EXPGRID workshop. Experimental Grid Testbeds for the Assessment of Large-Scale Distributed Apllications and Tools. In conjunction with HPDC-15*, pages 1–8. Paris, France, June 19th 2006.

[59] E. Caron and F. 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.

[60] E. Caron, F. Desprez, and G. Le Mahec. Dagda: An Advanced Data Manager for the Diet Middleware. In *AHEMA 2008: Proceedings of the Advances in High-Performance E-Science Middleware and Applications Workshop*. Indianapolis, Indiana USA, December 2008.

[61] E. Caron, F. Desprez, D. Loureiro, and A. Muresan. Cloud Computing Resource Management through a Grid Middleware: A Case Study with Diet and Eucalyptus. In IEEE, editor, *CLOUD 2009: IEEE International Conference on Cloud Computing*. Bangalore, India, September 2009. Published In the Work-in-Progress Track from the CLOUD-II 2009 Research Track.

[62] H. Casanova and F. Berman. *Grid Computing: Making The Global Infrastructure a Reality*, chapter Parameter Sweeps on the Grid with APST. John Wiley, 2003. Hey, A. and Berman, F. and Fox, G., editors.

[63] H. Casanova and J. 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. IEEE Computer Society, Washington, DC, USA, 1996. ISBN 0-89791-854-1.

[64] H. Casanova, A. Legrand, and M. Quinson. SimGrid: a Generic Framework for Large-Scale Distributed Experiments. In *10th IEEE International Conference on Computer Modeling and Simulation*. 2008.

[65] H. Casanova, A. Legrand, and Y. Robert. *Parallel Algorithms*. Chapman & Hall, 2008.

[66] J. S. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle. Managing Energy and Server Resources in Hosting Centers. *ACM SIGOPS Operating Systems Review*, 35(5):103–116, 2001.

[67] C.-C. Chiang, H.-K. Wu, W. Liu, and M. Gerla. Routing in Clustered Multihop, Mobile Wireless Networks with Fading Channel. In *Proceedings of IEEE SICON*, volume 97, pages 197–211. 1997.

[68] P. K. Chouhan. *Automatic Deployment for Application Service Provider Environments*. Ph.D. thesis, Ecole Normale Supérieure de Lyon, 2006.

[69] L. Cudennec, G. Antoniu, and L. Bougé. CoRDAGe: Towards Transparent Management of Interactions Between Applications and Ressources. In *STHEC/ICS 2008*. Island of Kos, Aegean Sea, Greece, June 2008.

[70] A. K. Datta, S. Devismes, and L. L. Larmore. A Self-Stabilizing $O(n)$-Round k-Clustering Algorithm. In *SRDS 2009: 28th International Symposium on Reliable Distributed Systems*. Niagara Falls, New York, September 2009.

[71] A. K. Datta, L. L. Larmore, and P. Vemula. Self-Stabilizing Leader Election in Optimal Space. In S. Kuklarni and A. Schiper, editors, *SSS 2008: 10th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, volume 5340 of *Lecture Notes in Computer Science*, pages 109–123. Springer, Detroit, MI, November 2008.

[72] A. K. Datta, L. L. Larmore, and P. Vemula. A Self-Stabilizing O(k)-Time K-Clustering Algorithm. *Computer Journal*, 2008.

[73] M. Davis, G. Efstathiou, C. S. Frenk, and S. D. M. White. The Evolution of Large-Scale Structure in a Universe Dominated by Cold Dark Matter. *Astrophysical Journal*, 292:371–394, May 1985.

[74] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. A Supernodal Approach to Sparse Partial Pivoting. *SIAM Journal on Matrix Analysis and Applications*, 20(3):720–755, 1999.

[75] A. Denis, C. Pérez, and T. Priol. PadicoTM: An Open Integration Framework for Communication Middleware and Runtimes. *Future Generation Computer Systems*, 19(4):575–585, May 2003.

[76] J. S. Deogun, D. Kratsch, and G. Steiner. An Approximation Algorithm for Clustering Graphs with Dominating Diametral Path. *Information Processing Letters*, 61(3):121 – 127, 1997.

[77] E. W. Dijkstra. Self-Stabilizing Systems in Spite of Distributed Control. *Communications of the ACM*, 17(11):643–644, 1974.

[78] S. Dolev. *Self-Stabilization*. MIT Press, Cambridge, MA, USA, 2000. ISBN 0-262-04178-2.

[79] S. Dolev, A. Israeli, and S. Moran. Self-Stabilization of Dynamic Systems Assuming only Read/Write Atomicity. *Distributed Computing*, 7(1):3–16, 1993.

[80] S. Dolev, A. Israeli, and S. Moran. Uniform Dynamic Self-Stabilizing Leader Election. *IEEE Transactions on Parallel and Distributed Systems*, 8(4):424–440, 1997.

[81] S. Dolev and N. Tzachar. Empire of Colonies: Self-Stabilizing and Self-Organizing Distributed Algorithm. *Theoretical Computer Science*, 410(6-7):514–532, 2009.

[82] C. Dominguez, R. Boelens, and A. Bonvin. HADDOCK: a Protein-Protein Docking Approach Based on Biochemical or Biophysical Data. *NMR-based docking of protein-protein complexes*, 125:51, 2003.

[83] J. Dongarra *et al.* Basic Linear Algebra Subprograms Technical Forum Standard. *International Journal of High Performance Applications and Supercomputing*, 16(1):1–111, 2002.

[84] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

[85] K. K. Droegemeier, D. Gannon, D. Reed, B. Plale, J. Alameda, T. Baltzer, K. Brewster, R. Clark, B. Domenico, S. Graves, E. Joseph, D. Murray, R. Ramachandran, M. Ramamurthy, L. Ramakrishnan, J. A. Rushing, D. Weber, R. Wilhelmson, A. Wilson, M. Xue, and S. Yalda. Service-Oriented Environments for Dynamically Interacting with Mesoscale Weather. *Computing in Science and Engineering*, 7(6):12–29, 2005.

[86] P. Eerola, B. Kónya, O. Smirnova, T. Ekelöf, M. Ellert, J. R. Hansen, J. L. Nielsen, A. Wäänänen, A. Konstantinov, J. Herrala, M. Tuisku, T. Myklebust, F. Ould-Saada, and B. Vinter. The NorduGrid Production Grid Infrastructure, Status and Plans. In *GRID'03: Proceedings of the 4th IEEE/ACM International Workshop on Grid Computing*, page 158. IEEE Computer Society, Washington, DC, USA, 2003. ISBN 0-7695-2026-X.

[87] A. Ephremides, J. E. Wieseltheir, and D. J. Baker. A Design Concept for Reliable Mobile Radio Networks with Frequency Hopping Signaling. 1987.

[88] D. G. Feitelson and L. Rudolph. Metrics and Benchmarking for Parallel Job Scheduling. In *Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lecture Notes in Computer Science*, pages 1–24. 1998.

[89] Y. Fernandess and D. Malkhi. K-Clustering in Wireless Ad Hoc Networks. In *POMC 2002: ACM Workshop on Principles of Mobile Computing*, pages 31–37. 2002.

[90] D. Fixsen, E. Cheng, J. Gales, J. Mather, R. Shafer, and E. Wright. The Cosmic Microwave Background Spectrum from the Full COBE FIRAS Data Set. *The Astrophysical Journal*, 473(2):576–587, 1996.

[91] A. Flissi, J. Dubus, N. Dolet, and P. Merle. Deploying on the Grid with DeployWare. In *CCGRID'08: Proceedings of the 8th IEEE/ACM International Symposium on Cluster Computing and the Grid*. Lyon, France, May 2008.

[92] A. Flissi and P. Merle. A Generic Deployment Framework for Grid Computing and Distributed Applications. In *GADA'06: Proceedings of the 2nd International OTM Symposium on Grid computing, High-PerformAnce and Distributed Applications*, volume 4279 of *Lecture Notes in Computer Science*, pages 1402–1411. Springer-Verlag, Montpellier, France, November 2006.

[93] I. Foster. Globus Toolkit Version 4: Software for Service-Oriented Systems. *Journal of Computer Science and Technology*, 21(4):513–520, 07 2006.

[94] I. Foster and A. Iamnitchi. On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing. *Peer-to-Peer Systems II*, pages 118–128, 2003.

[95] I. Foster and C. Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003. ISBN 1558609334.

[96] I. Foster, Y. Zhao, I. Raicu, and S. Lu. Cloud Computing and Grid Computing 360-Degree Compared. In *2008 Grid Computing Environments Workshop*, pages 1–10. IEEE, November 2008.

[97] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979. ISBN 0716710447.

[98] M. Gerla, G. Pei, and S.-J. Lee. Wireless, Mobile Ad-Hoc Network Routing. *Proceedings of the IEEE/ACM FOCUS'99*, 1999.

[99] S. Ghiasi, A. Srivastava, X. Yang, and M. Sarrafzadeh. Optimal Energy Aware Clustering in Sensor Networks. *Sensors*, 2(7):258–269, 2002.

[100] C. Gkantsidis, M. Mihail, and E. Zegura. Spectral Analysis of Internet Topologies. In *INFOCOM 2003: 23rd Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 364–374. 2003.

[101] A. S. Gokhale and D. C. Schmidt. Measuring and Optimizing CORBA Latency and Scalability Over High-Speed Networks. *IEEE Transactions on Computers*, 47:391–413, 1998.

[102] A. S. Gokhale and D. C. Schmidt. Principles for Optimizing CORBA Internet Inter-ORB Protocol Performance. In *HICSS'98: Proceedings of the Thirty-First Annual Hawaii International Conference on System Sciences-Volume 7*, page 376. IEEE Computer Society, Washington, DC, USA, 1998. ISBN 0-8186-8251-5.

[103] D. Grisby. omniORB, 2010. http://omniorb.sourceforge.net.

[104] S. Guha and S. Khuller. Approximation Algorithms for Connected Dominating Sets. *Algorithmica*, 20(4):374–387, 04 1998.

[105] D. Hagimont, P. Stolf, L. Broto, and N. Palma. Component-Based Autonomic Management for Legacy Software. *Autonomic Computing and Networking*, pages 83–104, 2009.

[106] S. Hatton, J. E. G. Devriendt, S. Ninin, F. R. Bouchet, B. Guiderdoni, and D. Vibert. GaLICS I: A Hybrid N-body Semi-Analytic Model of Hierarchical Galaxy Formation. *Monthly Notices of the Royal Astronomical Society*, 343:75–106, 2003.

[107] J. Holland. *Adaptation in Natural and Artificial Systems*. MIT press Cambridge, MA, 1992.

[108] Z. Hou, J. Tie, X. Zhou, I. Foster, and M. Wilde. ADEM: Automating Deployment and Management of Application Software on the Open Science Grid. In IEEE, editor, *Grid 2009, 10th IEEE/ACM International Conference on Grid Computing*, pages 130–137. IEEE, Banff, October 13-15 2009.

[109] A. Iosup, C. Dumitrescu, D. Epema, H. Li, and L. Wolters. How are Real Grids Used? The Analysis of Four Grid Traces and Its Implications. In *GRID'06: Proceedings of the 7th IEEE/ACM International Conference on Grid Computing*, pages 262–269. IEEE Computer Society, Washington, DC, USA, 2006. ISBN 1-4244-0343-X.

[110] P. Jetley, F. Gioachin, C. Mendes, L. V. Kalé, and T. Quinn. Massively Parallel Cosmological Simulations with ChaNGa. In IEEE, editor, *IPDPS'08: Proceedings of the 22th International Parallel and Distributed Processing Symposium*. Miami, Florida, April 2008.

[111] C. Johnen and L. H. Nguyen. Self-stabilizing Weight-Based Clustering Algorithm for Ad Hoc Sensor Networks. In S. E. Nikoletseas and J. D. P. Rolim, editors, *ALGOSENSORS 2006: Algorithmic Aspects of Wireless Sensor Networks, Second International Workshop, Venice, Italy, July 15, 2006, Revised Selected Papers*, volume 4240 of *Lecture Notes in Computer Science*, pages 83–94. Springer, 2006. ISBN 3-540-69085-9.

[112] C. Johnen and L. H. Nguyen. Robust Self-Stabilizing Weight-Based Clustering Algorithm. *Theoretical Computer Science*, 410(6-7):581–594, 2009.

[113] B. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *Bell System Technical Journal*, 49(2):291–307, 1970.

[114] T. Kichkaylo, A. Ivan, and V. Karamcheti. Sekitei: An AI Planner for Constrained Component Deployment in Wide-Area Networks. Technical Report TR2004-851, Department of Computer Science Courant Institute of Mathematical Sciences, New York University, March 2004.

[115] L. Kleinrock and F. Kamoun. Hierarchical Routing for Large Networks Performance Evaluation and Optimization. *Computer Networks (1976)*, 1(3):155 – 155, 1977.

[116] N. Krasnogor and J. Smith. MAFRA: A Java Memetic Algorithms Framework. In *Workshop Program, Proceedings of the 2000 Genetic and Evolutionary Computation Conference. Morgan Kaufmann*. 2000.

[117] B. Krishnamurthy and J. Wang. On Network-Aware Clustering of Web Clients. In *SIGCOMM'00: Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 97–110. ACM, New York, NY, USA, 2000. ISBN 1-58113-223-9.

[118] B. Krishnamurthy and J. Wang. Topology Modeling via Cluster Graphs. In *IMW'01: Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, pages 19–23. ACM, New York, NY, USA, 2001. ISBN 1-58113-435-5.

[119] S. Lacour. *Contribution à l'Automatisation du Déploiement d'Applications sur des Grilles de Calcul*. Thèse de doctorat, Université de Rennes 1, IRISA, Rennes, France, December 2005.

[120] S. Lacour, C. Pérez, and T. Priol. Generic Application Description Model: Toward Automatic Deployment of Applications on Computational Grids. In *GRID'05: Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*. Springer-Verlag, Seattle, WA, USA, November 2005.

[121] 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. D. 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.

[122] A. Legrand, A. Su, and F. Vivien. Minimizing the Stretch when Scheduling Flows of Biological Requests. In *SPAA'06: Proceedings of the 18th annual ACM symposium on Parallelism in algorithms and architectures*, pages 103–112. ACM Press, Cambridge, Massachusetts, USA, 2006. ISBN 1-59593-452-9.

[123] X. Li, B. Veeravalli, and C. Ko. Distributed Image Processing on a Network of Workstations. *International Journal of Computers and Applications*, 25(2):1–10, 2003.

[124] L. MartinGarcia. TCPDUMP/TCPCAP Public Repository, 2010. http://www.tcpdump.org.

[125] S. Matsuoka, H. Nakada, M. Sato, and S. Sekiguchi. Design Issues of Network Enabled Server Systems for the Grid. *Grid Computing —GRID 2000*, pages 59–85, 2000.

[126] M. Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, Cambridge, MA, USA, 1998. ISBN 0262631857.

[127] N. Mitton, A. Busson, and E. Fleury. Self-Organization in Large Scale Ad Hoc Networks. In *MED-HOC-NET 2004: The Third Annual Mediterranean Ad Hoc Networking Workshop*, volume 4. June 2004.

[128] N. Mitton, E. Fleury, I. Guerin L., and S. Tixeuil. Self-Stabilization in Self-Organized Multihop Wireless Networks. In *ICDCSW'05: Proceedings of the Second International Workshop on Wireless Ad Hoc Networking (WWAN)*, pages 909–915. IEEE Computer Society, Washington, DC, USA, 2005. ISBN 0-7695-2328-5-09.

[129] J. P. Morrison, B. Clayton, D. A. Power, and A. Patil. WebCom-G: Grid Enabled Metacomputing. *Neural, Parallel and Scientific Computations*, 12(3):419–438, 2004.

[130] T. Ng and H. Zhang. Predicting Internet Network Distance with Coordinates-Based Approaches. *INFOCOM 2002: 21st Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, 1:170–179, 2002.

[131] M. L. Norman, G. L. Bryan, R. Harkness, J. Bordner, D. Reynolds, B. O'Shea, and R. Wagner. Simulating Cosmological Evolution with Enzo. *ArXiv e-prints*, May 2007.

[132] OAR Team. OAR, 2010. http://oar.imag.fr.

[133] T. M. Oinn, M. Addis, J. Ferris, D. Marvin, R. M. Greenwood, T. Carver, M. R. Pocock, A. Wipat, and P. Li. Taverna: a Tool for the Composition and Enactment of Bioinformatics Workflow. *Bioinformatics*, 20(17):3045–3054, November 2004.

[134] OMG. CORBA Component Model, v4.0, 2010. http://www.omg.org/technology/documents/formal/components.htm.

[135] S. Ostermann. tcptrace - Official Homepage, 2010. http://www.tcptrace.org.

[136] J. P. Ostriker and M. L. Norman. Cosmology of the Early Universe Viewed Through the New Infrastructure. *Communications of the ACM*, 40(11):84–94, 1997.

[137] OW2 Consortium. The Fractal Project, 2010. http://fractal.ow2.org.

[138] D. Peleg. *Distributed Computing: a Locality-Sensitive Approach.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000. ISBN 0-89871-464-8.

[139] F. Pellegrini and J. Roman. SCOTCH: A Software Package for Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs. *High-Performance Computing and Networking*, pages 493–498, 1996.

[140] A. Penzias and R. Wilson. A Measurement of Excess Antenna Temperature at 4080 Mc/s. *Astrophysical Journal*, 142:419–421, 1965.

[141] A. Petitet, R. Whaley, J. Dongarra, and A. Cleary. HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers, 2010. `http://www.netlib.org/benchmark/hpl`.

[142] P. Petr Tůma. CCPsuite, 2010. `http://d3s.mff.cuni.cz/~ceres/prj/CCPsuite`.

[143] J.-F. Pineau. *Communication-Aware Scheduling on Heterogeneous Master-Worker Platforms*. Ph.D. thesis, École Normale Supérieure de Lyon, 2008.

[144] K. v. d. Raadt, Y. Yang, and H. Casanova. Practical Divisible Load Scheduling on Grid Platforms with APST-DV. In *IPDPS'05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, page 29.2. IEEE Computer Society, Washington, DC, USA, 2005. ISBN 0-7695-2312-9.

[145] M. Sato, M. Hirano, Y. Tanaka, and S. Sekiguchi. OmniRPC: A Grid RPC Facility for Cluster and Global Computing in OpenMP. *OpenMP Shared Memory Parallel Programming*, pages 130–136, 2001.

[146] 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*. 2004.

[147] R. Sharrock, T. Monteil, P. Stolf, D. Hagimont, and L. Broto. Non-Intrusive Autonomic Approach with Self-Management Policies Applied to Legacy Infrastructures for Performance Improvements. *IJARAS: International Journal of Adaptive, Resilient and Autonomic Systems*, 2(2):1–20, 2010.

[148] G. Singh, E. Deelman, G. Mehta, K. Vahi, M.-H. Su, G. Berriman, J. Good, J. Jacob, D. Katz, A. Lazzarini, K. Blackburn, and S. Koranda. The Pegasus Portal: Web Based Grid Computing. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 680–686. ACM Press, New York, NY, USA, 2005. ISBN 1-58113-964-0.

[149] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA, 1998. ISBN 0262692155.

[150] M. Spohn and J. Garcia-Luna-Aceves. Bounded-Distance Multi-Clusterhead Formation in Wireless Ad Hoc Networks. *Ad Hoc Networks*, 5:504–530, 2004.

[151] V. Springel. The Cosmological Simulation Code GADGET-2. *Monthly Notices of the Royal Astronomical Society*, 364:1105–1134, 2005.

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

[153] SUN Microsystems. S3L Library, 2010. `http://dlc.sun.com/pdf/816-0653-10/816-0653-10.pdf`.

[154] Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura, and S. Matsuoka. Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing. *Journal of Grid Computing*, 1(1):41–51, 03 2003.

[155] Y. Tanaka, H. Takemiya, H. Nakada, and S. Sekiguchi. Design, Implementation and Performance Evaluation of GridRPC Programming Middleware for a Large-Scale Computational Grid. In *GRID'04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 298–305. IEEE Computer Society, Washington, DC, USA, 2004. ISBN 0-7695-2256-4.

[156] Q. Tang, S. K. S. Gupta, D. Stanzione, and P. Cayton. Thermal-Aware Task Scheduling to Minimize Energy Usage of Blade Server Based Datacenters. *dasc*, 00:195–202, 2006.

[157] 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.

[158] K. Taura and A. Chien. A Heuristic Algorithm for Mapping Communicating Tasks on Heterogeneous Resources. *HCW 2000: Proceedings of the 9th Heterogeneous Computing Workshop*, pages 102–115, 2000.

[159] G. Tel. *Introduction to Distributed Algorithms.* Cambridge University Press, New York, NY, USA, 2001. ISBN 0521794838.

[160] R. Teyssier. Cosmological Hydrodynamics with Adaptive Mesh Refinement. A New High Resolution Code Called RAMSES. *Astronomy and Astrophysics*, 385:337–364, 2002.

[161] H. Topcuouglu, S. Hariri, and M.-y. Wu. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, 2002.

[162] Y. Wang, A. Carzaniga, and A. L. Wolf. Four Enhancements to Automated Distributed System Experimentation Methods. In *ICSE'08: Proceedings of the 30th international conference on Software engineering*, pages 491–500. ACM, New York, NY, USA, 2008. ISBN 978-1-60558-079-1.

[163] Y. Wang, M. J. Rutherford, A. Carzaniga, and A. L. Wolf. Automating Experimentation on Distributed Testbeds. In *ASE 2005: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering.* Long Beach, CA, November 7-11 2005.

[164] R. C. Whaley and A. Petitet. Minimizing Development and Maintenance Costs in Supporting Persistently Optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, February 2005.

[165] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated Empirical Optimizations of Software and the ATLAS Project. *Parallel Computing*, 27(1-2):3 – 35, 2001.

[166] R. Wolski, N. T. Spring, and J. Hayes. The Network Weather Service: a Distributed Resource Performance Forecasting Service for Metacomputing. *Future Generation Computer Systems*, 15(5-6):757–768, 1999.

[167] J. Wu and H. Li. On Calculating Connected Dominating Set for Efficient Routing in Ad-Hoc Wireless Networks. In *DIALM'99: Proceedings of the 3rd international workshop on Discrete algorithms and methods for mobile computing and communications*, pages 7–14. ACM, New York, NY, USA, 1999. ISBN 1-58113-174-7.

[168] Q. Xu and J. Subhlok. Automatic Clustering of Grid Nodes. In *GRID'05: Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, pages 227–233. IEEE Computer Society, Washington, DC, USA, 2005. ISBN 0-7803-9492-5.

[169] Z. Xu, S. T. Hedetniemi, W. Goddard, and P. K. Srimani. A Synchronous Self-stabilizing Minimal Domination Protocol in an Arbitrary Network Graph. *IWDC 2003: Distributed Computing*, pages 832–832, 2003.

[170] A. YarKhan, J. Dongarra, and K. Seymour. GridSolve: The Evolution of A Network Enabled Solver. *Grid-Based Problem Solving Environments*, pages 215–224, 2007.

[171] D. Zhu, R. Melhem, and B. R. Childers. Scheduling with Dynamic Voltage/Speed Adjustment Using Slack Reclamation in Multiprocessor Real-Time Systems. *IEEE Transactions on Parallel and Distributed Systems*, 14:686–700, 2003.

Publications

The publications are listed in reverse chronological order.

International Journal Articles

[CDDL10] Eddy Caron, Ajoy K. Datta, Benjamin Depardon, and Lawrence L. Larmore. A self-stabilizing k-clustering algorithm for weighted graphs. *Journal of Parallel and Distributed Computing*, 70(11):1159–1173, Nov 2010.

[ABC$^+$08] Abelkader 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 and Computation: Practice and Experience*, 20(9):1051–1069, 2008.

International Conference and Workshop Articles

[CDD10a] Eddy Caron, Benjamin Depardon, and Frédéric Desprez. Modelization and performance evaluation of the diet middleware. In *ICPP 2010, 39th International Conference on Parallel Processing*, pages 375–384. San Diego, CA, September 13-16 2010.

[CDD10b] Eddy Caron, Benjamin Depardon, and Frédéric Desprez. Deployment of a hierarchical middleware. In LNCS, editor, *Euro-Par 2010*, volume 6271 Part I of *LNCS*, pages 343–354. Ischia - Naples, Italy, August 31, September 3 2010.

[CDDL09] Eddy Caron, Ajoy K. Datta, Benjamin Depardon, and Lawrence L. Larmore. A self-stabilizing k-clustering algorithm using an arbitrary metric. In *Euro-Par 2009*, volume LNCS 5704, pages 602–614. TU Delft, Delft, The Netherlands, August 25-28 2009.

[DCD$^+$09] Benjamin Depardon, Eddy Caron, Frédéric Desprez, Jérémy Blaizot, and Hélène Courtois. Cosmological simulations on a grid of computers. In J.M. Alimi and A. Füzfa, editors, *INVISIBLE UNIVERSE: Proceedings of the Conference*, volume 1241, pages 816–825. AIP, Paris, France, 2009. ISBN 978-0-7354-0789-3.

[CCC$^+$07] Yves Caniou, Eddy Caron, Hélène Courtois, Benjamin Depardon, and Romain Teyssier. Cosmological simulations using grid middleware. In *Fourth High-Performance Grid Computing Workshop (HPGC'07)*. IEEE, Long Beach, California, USA, March 26 2007.

[ABB$^+$06] Abelkader Amar, Raphaël Bolze, Aurélien Bouteiller, Pushpinder Kaur Chouhan, Andréea Chis, Yves Caniou, Eddy Caron, Holly Dail, Benjamin Depardon, Frédéric Desprez, Jean-Sébastien Gay, Gaël Le Mahec, and Alan Su. Diet: New developments and recent results. In Lehner et al. (Eds.), editor, *CoreGRID Workshop on Grid Middleware (in conjunction with EuroPar2006)*, number 4375 in LNCS, pages 150–170. Springer, Dresden, Germany, August 28-29 2006.

National Conferences Articles

[Dep09] Benjamin Depardon. Un algorithme auto-stabilisant pour le problème du k-partitionnement sur graphe pondéré. In 19^e *Rencontres Francophones du Parallélisme (RenPar'19)*. Toulouse, September 9-11 2009.

[Dep08] Benjamin Depardon. Déploiement générique d'applications sur plates-formes hétérogènes distribuées. In 18^{e} *Rencontres Francophones du Parallélisme (RenPar'18)*. Fribourg, Switzerland, February 11-13 2008.

Research reports

[RR_CDD10a] Eddy Caron, Benjamin Depardon, and Frédéric Desprez. Modelization for the Deployment of a Hierarchical Middleware on a Heterogeneous Platform. Research Report RR-7309, INRIA, 06 2010. Also available as LIP Research Report RRLIP2010-19.

[RR_CDD10b] Eddy Caron, Benjamin Depardon, and Frédéric Desprez. Modelization for the Deployment of a Hierarchical Middleware on a Homogeneous Platform. Research Report RR-7201, INRIA, 02 2010. Also available as LIP Research Report RRLIP2010-10.

[RR_CDDL09] Eddy Caron, Ajoy Datta, Benjamin Depardon, and Lawrence Larmore. A Self-Stabilizing K-Clustering Algorithm Using an Arbitrary Metric. Research Report RR-7146, INRIA, 2009. Also available as LIP Research Report RRLIP2009-33.

[RR_DCD^{+}09] Benjamin Depardon, Eddy Caron, Frédéric Desprez, Hélène Courtois, and Jérémy Blaizot. Cosmological Simulations on a Grid of Computers. Research Report RR-7093, INRIA, 2009. Also available as LIP Research Report RRLIP2009-31.

[RR_Dep08] Benjamin Depardon. Déploiement générique d'applications sur plates-formes hétérogènes distribuées. Research Report 6434, Institut National de Recherche en Informatique et en Automatique (INRIA), January 2008. Also available as LIP Research Report 2008-06.

[RR_CCD^{+}07] Yves Caniou, Eddy Caron, Benjamin Depardon, Hélène Courtois, and Romain Teyssier. Cosmological simulations using grid middleware. Technical Report 6139, Institut National de Recherche en Informatique et en Automatique (INRIA), March 2007. Also available as LIP Research Report 2007-11.

[RR_ABB^{+}06] Abelkader Amar, Raphaël Bolze, Aurélien Bouteiller, Andréea Chis, Yves Caniou, Eddy Caron, Pushpinder Kaur Chouhan, Gaël Le Mahec, Holly Dail, Benjamin Depardon, Frédéric Desprez, Jean-Sébastien Gay, and Alan Su. Diet: New developments and recent results. Research Report 6027, INRIA, 11 2006. Also available as LIP Research Report 2006-31.

Abstract:

The results presented in this thesis deal with the execution of applications on heterogeneous and distributed environments: computing grids. We study, from end-to-end, the process allowing users to execute complex scientific applications. The contributions of this work are thus manifold. 1) Hierarchical middleware deployment: we first present an execution model for hierarchical middleware. Then, based on this model, we present several heuristics to automatically determine the shape of the hierarchy that would best fit the users' needs, depending on the platform it is executed on. We evaluate the quality of the approach on a real platform using the DIET middleware. 2) Graph clustering: we propose a distributed and self-stabilizing algorithm for clustering weighted graphs. Clustering is done based on a distance metric between nodes: within each created cluster the nodes are no farther than a distance k from an elected leader in the cluster. 3) Scheduling: we study the scheduling of independent tasks under resources usage limitations. We define linear programs to solve this problem in two cases: when tasks arrive all at the same time, and when release dates are considered. 4) Cosmological simulations: we have studied the behavior of applications required to run cosmological simulations workflows. Then, based on the DIET grid middleware, we implemented a complete infrastructure allowing non-expert users to easily submit cosmological simulations on a computing grid.

Keywords:

Grid middleware, deployment, graph clustering, k-clustering, self-stabilization, scheduling, cosmological simulations.

Résumé :

Les travaux présentés dans cette thèse portent sur l'exécution d'applications sur les environnements hétérogènes et distribués que sont les grilles de calcul. Nous étudions de bout en bout le processus permettant à des utilisateurs d'exécuter des applications scientifiques complexes. Les contributions de cette thèse se situent donc à plusieurs niveaux. 1) Déploiement d'intergiciel hiérarchique : nous proposons dans un premier temps un modèle d'exécution pour les intergiciels hiérarchiques. À partir de ce modèle, nous présentons plusieurs heuristiques pour définir automatiquement la meilleure hiérarchie en fonction des exigences des utilisateurs et du type de plate-forme. Nous évaluons la qualité de ces heuristiques en conditions réelles avec l'intergiciel DIET. 2) Partitionnement de graphe : nous proposons un algorithme distribué et auto-stabilisant pour partitionner un graphe quelconque ayant des arêtes pondérées entre les nœuds. Le partitionnement est réalisé en fonction des distances pondérées entre les nœuds et forme des grappes au sein desquelles les nœuds sont à une distance maximale k d'un nœud élu dans la grappe. 3) Ordonnancement : nous étudions l'ordonnancement de tâches indépendantes sous des contraintes de limitation d'utilisation des ressources. Nous définissons des formulations en programme linéaire pour résoudre ce problème dans deux cas : lorsque les tâches arrivent toutes en même temps et lorsqu'elles ont des dates d'arrivée. 4) Simulations cosmologiques : nous avons étudié le comportement d'applications nécessaires à l'exécution de workflows de simulations cosmologiques. Puis, en se basant sur l'intergiciel de grille DIET, nous avons mis en place une infrastructure complète permettant à des utilisateurs non expérimentés de soumettre facilement des simulations cosmologiques sur une grille de calcul.

Mots-clés :

Intergiciel de grille, déploiement, partitionnement de graphes, k-clustering, auto-stabilisation, ordonnancement, simulations cosmologiques.