IMT Institute for Advanced Studies, Lucca

Lucca, Italy

Design and evaluation of scalable approaches to schedule a stream of batch jobs in large-scale grids

PhD Program in Computer Science and Engineering

XX Cycle

By

Marco Pasquali

2008

The dissertation of Marco Pasquali is approved.

Program Coordinator: Prof. Ugo Montanari, University of Pisa

Supervisor: Dr. Ranieri Baraglia, Information Science and Technologies Institute-Italian National Research Council Pisa

Tutor: Dr. Ranieri Baraglia, Information Science and Technologies Institute-Italian National Research Council Pisa

The dissertation of Marco Pasquali has been reviewed by:

Prof. Luděk Matyska, Masaryk University

Prof. Domenico Talia, Universit della Calabria

IMT Institute for Advanced Studies, Lucca

2008

Contents

List of Figures

List of Tables

Acknowledgements

First of all, I want to show my gratitude to Ranieri Baraglia for his support, and the advice he gave me during my Ph.D. study. I want to thanks both Sun Microsystems, and all the High Performance Computing Lab. people at CNR-Pisa. They involved me in an innovative project, where I could propose my ideas, and I could attain the results presented in this work. A special thanks go to Patrizio Dazzi, a good friend to which I shared ideas and troubles, and to my wife, who encourages me every days.

Vita

October 3, 1980	Born, Carrara (Tuscany), Italy
2002	Bachelor's Degree in Computer Science Final mark: 101/110 University of Pisa, Pisa, Italy
2004	Master Degree in Computer Science Final mark: 105/110 University of Pisa, Pisa, Italy
2005	Ph.D. Student in Computer Science and Engineering IMT Institute for Advanced Studies, Lucca
2005	Teaching Assistant Department of Computer Science University of Pisa, Pisa, Italy
2005	Graduate Fellow Department of Computer Science University of Pisa, Pisa, Italy
2006	Graduate Fellow Information Science and Technologies Institute (ISTI) Italian National Research Council (CNR), Pisa, Italy

Publications

1. M. Pasquali, P. Dazzi, A. Panciatici, R. Baraglia, "Self-Optimizing Classifiers: Formalization and Design Pattern". In Proceedings of the CoreGRID Symposium in conjunction with EuroPar 2008. Las Palmas de Gran Canaria, Canary Island, Spain. August, 2008.
2. A.D. Techiouba, G. Capannini, R. Baraglia, D. Puppin, M. Pasquali, L. Ricci, "Backfilling Strategies for Scheduling Streams of Jobs on Computational Farms". Making Grids Work, CoreGRID series, Springer Verlag. June, 2008.
3. M. Pasquali, R. Baraglia, G. Capannini, D. Laforenza, L. Ricci. "A two-level scheduler to dynamically schedule a stream of batch jobs in large-scale grids". HPDC ACM/IEEE International Symposium on High Performance Distributed Computing, Boston, US, June 23-27 2008. Poster.
4. D. Klusáček, H. Rudová, R. Baraglia, M. Pasquali, G. Capannini. "Comparison of multi criteria scheduling techniques". In CoreGRID Integration Workshop 2008. Integrated Research in Grid Computing. Heraklion-Crete, Greece, April 2-4, 2008.
5. R. Baraglia D. Puppin M. Pasquali L. Ricci A.D. Techiouba, G. Capannini. "Backfilling strategies for scheduling streams of jobs on computational farms". In CoreGRID Workshop on Grid Programming Model, Grid and P2P Systems Architecture, Grid Systems, Tools and Environments, June 12-13, 2007. Heraklion - Crete, Greece.
6. P. Dazzi, F. Nidito, and M. Pasquali. "New perspectives in autonomic design patterns for stream-classification-systems". In WRASQ '07: Proceedings of the 2007 workshop on Automating service quality, pages 34–37, New York, NY, USA, 2007. ACM.
7. D. Puppin L. Ricci M. Pasquali G. Capannini, R. Baraglia. "A job scheduling framework for large computing farms". In SC 07 ACM/IEEE Computer Society International Conference for High Performance Computing, Networking, Storage, and Analysis, 2007.
8. P. Dazzi, A. Panciatici, M. Pasquali. "A formal checklist for self-optimizing strategy-driven priority classification system". Technical Report 2008-TR-005, ISTI-CNR, February 2008.
9. P. Dazzi, M. Pasquali. "Formalization of autonomic heuristics-driven systems". Technical Report 2008-TR-004, ISTI-CNR, February 2008.

10. M. Coppola, M. Pasquali, L. Presti, M. Vanneschi, “An Experiment with High Performance Components for Grid Applications”. In IPDPS: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium 2005.

Presentations

1. "A two-level scheduler to dynamically schedule a stream of batch jobs in large-scale Grids". 7th Meeting of the Institute on Resource Management and Scheduling, CoreGrid WP6, March, 2008, Dortmund, Germany.
2. "New perspectives in autonomic design patterns for stream-classification-systems". In WRASQ '07: Proceedings of the 2007 workshop on Automating service quality: Held at the International Conference on Automated Software Engineering (ASE). Atlanta, Georgia, 2007.
3. "A Job Scheduling Framework for Large Computing Farms". International Research Workshop on Scheduling WS 07 Cetraro, Italy June 25th 29th, 2007.
4. "Backfilling strategies for scheduling streams of jobs on computational farms". In CoreGRID Workshop on Grid Programming Model, Grid and P2P Systems Architecture, Grid Systems, Tools and Environments, June 12-13, 2007. Heraklion - Crete, Greece.

Abstract

I think the grid computing stimulates the cooperation among people, that agree to share resources and knowledge, to address problems that are not reachable individually. The central purpose of grid computing is to enable a community of users to perform work on a pool of shared resources. Since the number of jobs to be done in most cases outnumbers the number of available resources, somebody must decide how to allocate resources to jobs. Historically, this has been known as the *Scheduling Problem*. A large amount of research in scheduling was motivated by the proliferation of massively parallel machines in the early 1990s, and the desire to use these very expensive resources as efficiently as possible. Nowadays, grid is the emerging computing platform. Grid computing makes the scheduling problem even more difficult since, due to its heterogeneous and highly distributed nature, it cannot be served by a single centralized scheduling framework.

In this Thesis we propose the study conducted to design and evaluate a hierarchical framework to dynamically schedule a continuous stream of independent, batch jobs, on large-scale grids. Our scheduler aims to schedule arriving jobs respecting their computational requirements (hardware constraints, software resources needed to be executed, deadlines), and optimizing the utilization of hardware and software resources. We designed a lightweight Meta-Scheduler able to classify incoming jobs according to their relevance, and to schedule jobs among underlying resources balancing the workload among them. Furthermore, we designed different algorithms as Local-Schedulers, which are able to carry out the job machine associations using dynamic information about the environment.

Eventually, the Thesis presents two important related results obtained by scientific collaborations. The former concerns to the study of the self-optimizing system behavior. The latter concerns to the comparison of one of our Local-Scheduler solution with a Schedule-Based algorithm proposed by Dalibor Klusáček and Hana Rudová of the University of Brno.

Chapter 1

Introduction

1.1 Motivations

The research activity described in this Thesis was conducted as part of the "*Scheduling Under the Sun*" project, that was born as a collaboration between the Sun Microsystems™ and the Italian National Research Council. The goal of this research is to design and evaluate a scalable solution to dynamically schedule a stream of batch jobs in large scale grids for utility computing. The objective of our work was to develop a scheduler able to manage the execution of a large number of jobs on a grid infrastructure composed by thousands machines.

The Sun Microsystems™ challenge is to provide its customers with a computational infrastructure able to run different kinds of high-performance, compute-intensive applications (72). This without the need for users to have their own infrastructure and, above all, their own schedulers.

The decreasing cost of the hardware resources, and their easy interconnection, allows to develop large computing farms, but this is not enough to provide computational services because, the increasing complexity of the hardware infrastructures requires that the complexity of the software infrastructures needed to manage those increases.

Some important questions that this scenario opens are: what machine

can be assigned to a job? How long? Does this assignment allow to maximize the resource utilization? Does it allow to satisfy the Quality of Service (QoS) required by applications?

In literature, a general form of this problem is known as the *Scheduling Problem*. A more formal definition of such problem is given in (80): a schedule of jobs is the assignment of jobs to specific time intervals of resources, such that no two jobs are on any resource at the same time, or such that the capacity of no resource is exceeded by jobs. The schedule of jobs is optimal if it minimizes/maximizes a given optimality criteria (objective function). A scheduling problem is specified by *a set of machines*, *a set of jobs*, *one or more optimality criteria*, *environmental specifications*, and by *other constraints*. The scheduling problem solutions goal is to find an optimal schedule in the environment that satisfies all constraints. The scheduling problem has shown to be NP-complete in its general as well as in some restricted forms (57).

The structure of a scheduler depends on the features and number of resources managed, and the domain in which resources are located. We can distinguish three different scheduling architectures: centralized, distributed, and hierarchical (18; 60).

The centralized can be used for managing single or multiple resources located either in a single or in multiple domains. It can only support a uniform policy and suits well for cluster management systems such as Condor (70), LSF (6), and Condine (49). It is not suitable for grid resource management systems as they are expected to respect (local) policies imposed by resource owners.

In the distributed model, schedulers interact among themselves in order to decide which resources should be assigned to the jobs to be executed. Condor-G (42) is an example of a distributed scheduler. It is an extension of Condor able to manage different instances spread in different domains using the Globus support (40). In this scheme, there is not a central leader responsible for scheduling, hence the model is potentially highly scalable and fault-tolerant. As resource owners can define the policy that schedulers can enforce. However, because the status of remote jobs and resources are not available in a single location, the generation of

highly optimal schedule is not guaranteed. The distributed scheme can suit grid systems, but, in general, it is difficult because resource owners do not always agree on a global policy for resource management.

The hierarchical model is suitable for grid environments. In literature there are several hierarchical schedulers, such as YML (31) (75), OAR (23), KOALA (74). In the next Chapter a survey of such systems is given. This model can be seen as a hybrid one, a combination of the central and the distributed models. It allows remote resource owners to enforce their own policy on external users at local level. Furthermore, it defines a high level scheduler able to orchestrate the interactions among local scheduler instances. The hierarchical model overcomes the most important limitations of the distributed model, preserving a high level of scalability. The scheduling framework investigated in this Ph.D study follows this model (85).

1.2 Problem Description

Referring to the definition of the scheduling problem given in the previous section, we present the work done during our research activity as a solution for a restricted instance of the general problem (80).

The aim of this Ph.D study is to investigate a solution to design a scheduler able to manage a set of distributed and heterogeneous resources, single-processor or SMP machines, organized as clusters located in specific sites (*a set of machines*). The target computing grid is a dedicated one, able to notify configuration changes such as job submission/ending. The links among machines belonging to the same cluster are high-bandwidth ones, while the clusters are connected by the internet infrastructure.

In our study we consider a continuous stream of batch jobs (*a set of jobs*). We suppose that a job is sequential or multi-threaded, that it is executed only on a single machine, that jobs are allocated to a machine according to the space sharing policy, and that jobs are independent, i.e. the execution of a job does not depend on the execution of other jobs. We also assume that some jobs are preemptable.

The jobs are managed according to the on-line paradigm. In (87), Sgall classifies the on-line algorithms according to which part of them is given on-line:

- Scheduling jobs one by one. This means that jobs arrive one at a time. As soon as the job arrives, the scheduler knows all its characteristics, including its running time. Each job has to be scheduled before the next job arrives. The scheduler cannot change the previous assignments.
- Unknown running times. At any time all currently available jobs are at the disposal of the scheduler. Any of them can be started suddenly or delayed further. The running time of a job is unknown until the job finishes. The scheduler only knows whether a job is still running or not. Furthermore, if preemption or restart are allowed, the scheduler can decide to preempt or stop any job which is currently running.
- Jobs arrive over time. In this paradigm, the algorithm has the same freedom as in the previous case: jobs are at the disposal of the scheduler and, they can be started suddenly or delayed further. In addition the running time of each job is also known when a job is available. The scheduler does not know the length of the job stream.
- Interval scheduling. Interval scheduling assumes that each job has to be executed in a precisely given time interval. If this is impossible, it may be rejected.

In our studies, the Meta-Scheduler dispatches jobs according to the "Jobs arrive over time" classification. We assume that users can submit their jobs at any time without any predefined policies. This implies that the scheduler doesn't know the length of the stream. For each submitted job its running time is specified, and jobs can be scheduled or delayed according to the scheduler's decisions. Moreover, the Meta-Scheduler cannot change its scheduling decision. This means that when a job is assigned to a Local-Scheduler, it cannot be migrated to another Local-Scheduler.

The scheduler aims to maximize the hardware/software resource usage, to schedule jobs among clusters balancing the workload, and to guarantee the QoS request by submitted jobs, e.g. job deadlines, software licences requirements, SLA of the submitting user (*one or more optimality criteria*).

During our studies we investigated a multi-level scheduler, and we specified what kind of information concerning resources are available at each level of the hierarchy (*environmental specifications*).

Eventually, we considered some constraints both about jobs, machines and software resources (*constraints*). Submitted jobs and machines are annotated with information describing computational requirements and hardware/software features, respectively. Each job is described by an identifier, its deadline, this means that jobs can specify a time at which their execution has to be finished, an estimation of job execution time, a benchmark score, which represents the architecture used to estimate its execution time, the number of processors and licences requested. Machines are described by a benchmark score, which specifies their computational power, the number of CPUs, and the types of different software licence that they are able to run. Concerning software licences, let L be the set of different software licences, each one with a specific number of maximum activable copies. Each job can require a subset $L_c \subseteq L$ of licences to be executed. At any time, the number of active licences must not be greater than their availability.

1.3 Contributions of the Thesis

Our aim was to design and to evaluate a scalable job scheduling framework (85). The main features of the proposed scheduler concern with the ability to schedule jobs among distributed resources, taking care of a set of different, and sometimes in contrast, constraints. In particular, we define a job scheduler able to manage software resources needed to execute jobs, such as licences, and to address the job deadline requirement.

The first step was to choose the most profitable architecture for the scheduler we were designing. We adopted a hierarchical model with in

mind to design a lightweight Meta-Scheduler, that works as single access point to the infrastructure (Section 3.1), and to design a Local-Scheduler able to carry out the job machine associations using dynamic information about the environment (Section 3.2).

Another important aspect was to define a validation model for our framework. In a real grid environment, it is hard, and perhaps even impossible, to perform scheduler performance evaluation in a repeatable and controllable manner for different scenarios. The availability of resources and their load continuously varies with time, and it is hard to control the activities of users in different administrative domains. There are two common ways to evaluate a system design (33): using real traced workload, or creating a model from the trace and use the model for either analysis or simulation. The advantage of using a trace directly is that it is the most "real" test of the system; the workload reflects a real workload precisely, with all its complexities, even if they are not known to the person performing the analysis. The drawback is that the trace reflects a specific workload, and there is always the question of whether the results generalize to other systems. In particular, there are cases where the workload depends on the system configuration, and therefore a given workload is not necessarily representative of workloads on systems with other configurations. This makes the comparison of different configurations problematic. In addition, traces are often misleading if we have incomplete information about the circumstances when they were collected. For example, workload traces often contain intervals when the machine was down or part of it was dedicated to a specific project, but this information may not be available.

We studied the real workload traces proposed by Feitelson in (36), and the data provided by the Italian Interuniversity Consortium (25). In both cases real workload traces missing requirements fundamental in the scheduling framework we propose, i.e. deadlines and software requirements. Furthermore, the systems from which the traces come from are different and incompatible with our target computing platform.

We choose to evaluate our system by simulations, using different streams of jobs synthetically generated with different inter-arrival times

between jobs. To slacken the job estimation execution time problem, we assume for each job the estimated execution time is accurate, and that any job cannot run longer than its estimation. Most of the common scheduling algorithms and frameworks, included those proposed in this Thesis, take their scheduling decisions assuming that the job running time prediction is enough precise. They exploit the job estimation execution time to build scheduling plans, and to achieve their goals. In literature many methods have been proposed to estimate the job execution time (46; 88). Usually, they are based on system historical data analysis, e.g. they operates on the principle that applications with similar characteristics have similar execution times (64; 92). The main problem of these approaches is the definition of similarity: different criteria can be adopted to define two applications similar.

We developed an ad-hoc simulator for our evaluation (43). We simulated a computing farm, with varying number of jobs, machines and licences. Each machine in the simulator was randomly generated. A simulation step includes: (1) selection of new jobs, (2) update of the status (e.g. the job execution progress) of the executing jobs, (3) check for job terminations.

In Chapter 4, we show that in the conducted evaluations, the scheduling framework we propose is able to balance the workload among clusters, and to maximize the number of jobs executed respecting their QoS requirements.

The Meta-Scheduler level

The Meta-Scheduler we proposed has a twofold goal:

- to assign each job to the underlying schedulers in such a way that the workload is balanced among clusters, and that the higher priority jobs could be executed before than the lower ones.

- to assign to each job a priority value, with respect to its QoS requirements, needed to define a job execution order.

In literature, we can find different approaches that exploit a hierarchical architecture for job schedulers, which prioritizes incoming jobs, and that have as goal to balance the resources workload (5; 23; 31; 74; 75; 91). These approaches have a light and static jobs classification, not extensible to new jobs QoS requirements or administrators' policies. In most cases, these classifications are based only on priority queues mechanisms. Our approach allows to define an extensible set of classification heuristics, each one managing a specific constraint. The most of the analyzed schedulers (Chapter 2) use negotiation mechanisms to plan the job schedule. This introduces an overhead due to communication among scheduler levels. Our Meta-Scheduler takes the scheduling decisions analyzing only the Local-Scheduler queues. This means that Local-Schedulers need to communicate to the high level only the termination of assigned the jobs (Section 3.1). As shown in the experiments (Chapter 4), this choice allows to achieve a good workload balancing among clusters reducing the communication costs between levels.

The Meta-Scheduler is characterized by two phases: the first one is the job classification, in which each job is characterized by a priority value, the second one is the veritable job scheduling among the Local-Schedulers.

The job classification phase is based on a set of heuristics, each one able to manage a specific problem constraint. In particular we identify three different job parameters interesting for the classification: deadline, requested licences and user peculiarities. Each heuristics considers a specific job parameter to compute a value that contributes to fix the job priority. When all the contributions are carried out, they are combined to obtain the job priority. What we want to remark as new, it is that each heuristics works without any knowledge of the resources status, according to the on-line paradigm, and that the heuristics set can be extended to consider other new QoS requirements. Experiments (Chapter 4) show that the classification phase allows to improve the number of jobs that are executed respecting their QoS requirements. Furthermore, the con-

ducted tests show that the results carried out by this phase are comparable with the results carried out by more complex classification strategies, with smaller computational cost.

Moreover, we studied how to apply autonomic features to our classification mechanism in such a way that it adapts its behavior to the operating environment (Section 5.1). In literature there are not many efforts following this direction. In (81), Pendarakis et al. apply autonomic concept to overcome the inability of schedulers to manage networks to reach the desired resource utilization. They present an autonomic bandwidth control system that adaptively adjusts incoming and outgoing traffic rates to achieve system management goals. Respect to the autonomic bandwidth control system, we follow another direction to exploit autonomic features in our scheduler. Our effort is to apply the design pattern for autonomic classifiers (30) to the Meta-Scheduler classification phase. Here, the objective is to design our classifier in such a way it is able to classify incoming jobs according to a given priority distribution policy, and to adapt its behavior to respect the given policy instead of system changes (Section 5.1).

Once a job is labeled with a priority in the classification phase, it is elaborated by the scheduling phase of our Meta-Scheduler. In this phase the priority values is used to schedule jobs balancing the workload among the underlying level. To implement the scheduling phase, we investigated two scheduling functions: *Load* and *Ordering*. Load aims to balance the workload among clusters by assigning a job to the less loaded cluster. Ordering aims to balance the number of jobs with equal priority in each cluster queue. We designed this phase in such a way it makes easy to extend the set of scheduling functions in order to provide for new policies (e.g. changing the optimality criterion in minimizing the waiting time of the queued jobs). As shown in the experiments (Chapter 4), the results obtained using these functions point out that the workload assigned to each cluster is proportional to its estimated computational power.

The Local-Scheduler level

We investigated different solutions at Local-Scheduler level: a Flexible Backfilling algorithm able to manage our specific constraints (35), a new solution based on the Convergent Scheduling technique (43).

As the Meta-Scheduler, the solution based on the Flexible Backfilling algorithm at Local-Scheduler level is composed by the classification and scheduling phases (Section 3.2.1). The classification phase assigns to each job a priority value computing a set of heuristics. Each heuristics manages a specific problem constraint, but, unlike the Meta-Scheduler, it exploits resource information, machines status and licences availability, and it knows features about running and queued jobs. This allows the Local-Scheduler to compute more accurate priorities than those computed of Meta-Scheduler level, but this also implies that the complexity of the Local-Scheduler classifier is greater than that of the Meta-Scheduler. The heuristics set designed and implemented for this phase constitutes an innovative step cornering the job classification, when the objectives are to respect the job deadlines, and to maximize the hardware and software resource usage.

The proposed Convergent Scheduling technique (43) permits us to carry out a job-scheduling plan on the basis of the current status of the system (i.e. resource availability, executing jobs), and information related to jobs waiting for execution (Section 3.2.2). In order to take decisions, the scheduler assigns priorities to all the jobs in the cluster, and job priorities are computed at each scheduling event. Job priorities measure the degree of preference of a job for a cluster machine, i.e. how the machine suits the job execution. Jobs are labeled with a priority value using a set of heuristics. Each heuristics increases/decreases the degree of the matching between a job and a machine. Heuristics can be run in any order, and each heuristics manages a specific problem constraint. The scheduler aims to schedule a subset of queued jobs that maximize the degree of preference for the available resources, and that can be simultaneously

executed without violating the constraints on licence usage.

1.4 Thesis Organization

This Thesis is structured as follows. In Chapter 2 is presented a survey of the current state of the art in the field of job scheduling algorithms and frameworks. Chapter 3 is devoted to the explanation of the novel solution we proposed. We will describe the multi-level architecture we designed and implement, and we will point out the main aspects of the proposed Meta-Scheduler, and of the proposed algorithms used at the local Scheduler level. In Chapter 4, we present some results carried out by the experimental phases. We will show the results obtained by the Flexible Backfilling and the Convergent Scheduling algorithms. Furthermore, we will point out the results of the interaction between the Meta-Scheduler and the two algorithms used at Local-Schedulers level. In Chapter 5 we describe some related results obtained during our research activity, and by the collaboration with the university of Brno. We will present a design pattern for autonomic Stream-Classification-Systems we defined, and how we applied the pattern to the implementation of the autonomic classifier used at the Meta-Scheduler level. Moreover, we present the scheduler proposed by Klusáček and Rudová, and the comparison with our Flexible Backfilling. In the last Chapter, we present the conclusive considerations of our work and we describe the future works concerning both levels of the hierarchical architecture we proposed.

Chapter 2

State of Art

Grids are geographically distributed platforms composed of heterogeneous resources that users can access via a single interface, providing common resource-access technology and operational services across widely distributed and dynamic virtual organizations, i.e. institutions or individuals that share resources. Resources are generally meant as reusable entities employable to execute applications, and comprise processing units, secondary storage, network links, software, data, special-purpose devices, etc. Grid computing differs from conventional distributed computing as it focuses on large-scale coordinated resource sharing by independent providers in different administrative domains. Grids are conceived to support innovative applications in many fields, and they are today mainly used as effective infrastructures for distributed high-performance computing and data processing. However, their application areas are shifting from scientific computing towards industry and business-related applications. Grid resource management and scheduling components are important for building grids. They are responsible for the selection and allocation of grid resources to current and future applications. Thus, they are the building blocks for making grids available to user communities (14; 50; 61).

A grid can be seen as a seamless, integrated computational and collaborative environment. One of the user's access points to a grid can be a

Grid Resource Broker (GRB). Users interact with it to submit and execute jobs. The GRB performs resource discovery, scheduling, and the managing of application jobs on distributed grid resources. Grid computing makes the scheduling problem even more difficult since, due to its heterogeneous and highly distributed nature, a single centralized scheduler algorithm is not suitable for grid resource management systems.

The overall aim of a GRB is to efficiently and effectively schedule applications that need to utilize the available resources. From the user point of view, resource management should be transparent. A Grid Resource Management System (RMS) consists of many vital components (22; 37). The RMS, and in particular the scheduler, plays a major role when issues like acceptance, usability, or performance of a machine are considered. The objective of the scheduler is twofold: to maximize the overall resource utilization guarantying the required applications performance.

In this chapter we survey the main scheduling algorithms and frameworks, which can be found in literature. Different classification can be made when schedulers are analyzed (24; 39): according to their architecture (e.g. centralized, distributed, and hierarchical), according to their behavior (e.g. static and dynamic), according to the way they elaborate the incoming jobs (e.g. online and off-line), according to the resulting schedule (e.g. queue-based and schedule-based). In this Thesis, we propose a scheduler that can be classified as hierarchical, and it processes the incoming jobs according to the online paradigm. Moreover, our scheduler can be classified as static, this means that jobs are assigned to the appropriate resources before their execution begins. Once started, they run on the same resources without interruption. Opposed to static, dynamic algorithms allow the reevaluation of already determined assignment decisions during job execution. Dynamic schedulers can trigger job migration or interruption based on dynamic information about both the status of the system, and the application workload. Concerning the resulting schedules, we design a queue-based scheduling framework, this means that the scheduler, both at Meta as at Local level, takes as input jobs stored in a queue, and it tries to assign them to the available resource. In Chapter 5, we compare our solution to a schedule-based algorithm that

compute a scheduling plan in which it is know when each job in the system will start/end its execution.

In the past many job-scheduling algorithms and frameworks have been proposed (13; 24; 38; 45; 63; 69), some schedule jobs storing them in different queues (4), which are served using different policies, but they don't characterize jobs with a priority value. Furthermore, any of the analyzed frameworks consider software resources that can be required by jobs to be executed, and only Nimrod-G allows users to specify a deadline for their jobs (8).

In the first part of this chapter, we will describe some scheduling algorithms that are exploited in different schedulers. In particular, we will focus on the Backfilling algorithm (35) because it is employed in the most common scheduling frameworks, and we also exploit it in our solution. Backfilling can be described as centralized scheduler, but it is employed in many scheduling frameworks either they have a different architecture. This because if we see only to the a restricted set of resources, which need to be handled in an autonomous way, we can employ a specific scheduling algorithm (i.e. the local scheduler in our solution), in this way the Backfilling can co-exit in a distributed and/or hierarchical schedulers.

In the second part of the Chapter, we will describe some scheduling frameworks, pointing out the main difference of each of them with our approach. All of the analyzed schedulers works according to the queue-based approach and according to the on-line paradigm, we will specify when dynamic mechanisms are available.

2.1 Scheduling Algorithms

2.1.1 List scheduling

The *list scheduling* (LS) algorithm was originally proposed by Graham (1969) (67) for scheduling sequential jobs that demand for only one processor. A list schedule is based on an initial ordering of the jobs L, called a priority list. Initially, at time zero, the scheduler instantaneously scans the list L from the beginning, searching for jobs that are ready to be ex-

ecuted, i.e., which have no predecessors jobs still waiting in L. The first ready jobs in L is removed from the list, and sent to an idle processor for processing. Such a search is repeated until there is no ready job, or there is no more processors available. In general, whenever a processor completes a job, the scheduler immediately scans L, looking for the first ready job to be executed. If such a ready job is not found, the processor becomes idle, and waits for the next finished job.

As running jobs complete, more precedence constraints are removed, and more jobs will be ready. For a given computation C, the length of a list schedule certainly depends on the priority list L. However, the performance of an LS schedule is quite robust. Graham proved that the list schedule length is no more than twice the optimal schedule length for sequential jobs.

The excellent performance of the list scheduling algorithm motivates researchers to apply the strategy to scheduling parallel jobs on multiprocessors. The extension of the method to parallel jobs is straightforward. When the scheduler finds a ready job t that need p processors to be executed; the scheduler checks whether there are at least p idle processors. If so, the job t is allocated on p processors and executed non-preemptively on these processors. Otherwise, the ready job t still needs to wait in L until other running jobs complete. Therefore, in addition to precedence constraints; there are also processor constraints; which make the problem more complicated than the one solved by Graham.

In the presented list scheduling algorithm it is assumed a noncontiguous processor allocation scheme, i.e., if there are at least p idle processors, then any p processors can be allocated to the job t. This assumption simplifies the scheduling problem. In general, it is not true that all processors can execute all jobs in L. In particular, given a priority list L, an LS schedule is not unique, and different processor allocation strategies result in different schedule lengths. Finding an optimal LS schedule for a given list L such that the resulting schedule is minimized is a nontrivial problem.

2.1.2 First Come First Served

In this section, we present *First-Come-First-Served* (FCFS) scheduling algorithm, a simple job scheduling method that is a specialization of the above list scheduling (86). The FCFS algorithm schedules jobs with respect to their submission order. If not enough resources are available to schedule the first queued job, FCFS waits until the required resources become free, and the job can be scheduled.

2.1.3 Backfilling

Backfilling is an optimization of the *First-Come-First-Served* algorithm (FCFS) (86). Backfilling tries to balance the goals of resource utilization, maintaining the FCFS jobs order (35). As matter of fact, FCFS guarantees that jobs are started in the order of their arrivals, the same property does not necessarily hold for the completion of the jobs due to differences in job execution times. This implies that FCFS is fair in the formal sense of fairness, but it is unfair in the sense that long jobs make short jobs wait and unimportant jobs make important jobs wait. Backfilling introduces an optimization to the FCFS algorithm in such a way that the completion time of any job does not depend on any other job submitted after it.

Backfilling requires that each job specifies its estimated execution time. While the job at the head of the queue is waiting, it is possible for other, smaller jobs, to be scheduled, especially if they would not delay the start of the job on the head of the queue. Processors get to be used that would otherwise remain idle. By letting some jobs execute out of order, other jobs may get delayed. Backfilling will never completely violate the FCFS order where some jobs are never run (a phenomenon known as *starvation*). In particular, jobs that need to wait are typically given a reservation for some future time.

Backfilling, in which small jobs move forward to utilize the idle resources, was introduced by Lifka (69). This was done in the context of EASY, the Extensible Argonne Scheduling sYstem, which was developed for the first large IBM SP1 installation at Argonne National Lab.

While the concept of backfilling is quite simple, it nevertheless has

several variants with subtle differences. It is possible to generalize the behavior of backfilling by parameterizing several constants. Judicial choice of parameter values lead to improved performance. One parameter is the *number of reservations*. In the original EASY backfilling algorithm, only the first queued job received a reservation. Jobs may be scheduled out of order only if they do not delay the job at the head of the queue. The scheduler estimates when a sufficient number of processors will be available for that job and reserves them for this job. Other backfilled jobs may not violate this reservation, they must either terminate before the time of the reservation, or use only processors that are not required by the first job .

Backfilling may cause delays in the execution of other waiting jobs (which are not the first, and therefore do not get a reservation). The obvious alternative is to make reservations for all queued jobs. This approach has been named *Conservative Backfilling*. Simulation results indicate, however, that delaying other jobs is rarely a problem, and that conservative backfilling tends to achieve reduced performance in comparison with the more aggressive EASY backfilling (35).

A recent suggestion is adaptive reservations depending on the extent different jobs have been delayed by previous backfilling decisions. If a job is delayed by too much, a reservation is made for this job. This is essentially equivalent to the earlier conservative backfilling, in which all queued jobs have reservations, but, in this case, it is allowed to violate these reservations up to a certain slack. Setting the slack to the threshold used by adaptive reservations is equivalent to only making a reservation if the delay exceeds this threshold. Another parameter is the *order of queued jobs*. The original EASY scheduler, and many other systems and designs, use a FCFS order. A general alternative is to prioritize jobs in some way, and select jobs for scheduling (including candidates for backfilling) according to this priority order. *Flexible backfilling* combines three types of priorities: an administrative priority set to favor certain users or projects, a user priority used to differentiate among the jobs of the same user, and a scheduler priority used to guarantee that no job is starved.

In general, job priorities are computed as a function of the job charac-

teristics. In particular, Chiang et al. have proposed a whole set of criteria based on resource consumption, that are generalizations of the well-known *Shortest Job First* (SJF) scheduling algorithm (35). These have been shown to improve performance metrics, especially those that are particularly sensitive to the performance of short jobs, such as slowdown. Another parameter is the amount of lookahead into the queue. All previous backfilling algorithms consider the queued jobs one at a time, and try to schedule them. But, the order in which jobs are scheduled may lead to loss of resources due to fragmentation. The alternative is to consider the whole queue at once, and try to find the set of jobs that together maximize the desired performance metrics. This can be done using dynamic programming, leading to optimal packing and improved performance.

In our study, we developed a flexible backfilling algorithm, which job queue is ordered exploiting a priority value computed by a set of heuristics for each incoming job (35). We used our flexible backfilling at the Local level of the hierarchical framework we proposed (85). Furthermore, we exploited our backfilling strategy to evaluate the feasibility of the Convergent Scheduler technique (43).

2.1.4 Gang Scheduling

In *Gang Scheduling* jobs can be preempted and re-scheduled as a unit, across all involved resources. The notion was introduced by Ousterhout (76), using the analogy of a working set of memory pages to argue that a working set of processes should be co-scheduled for the application to make efficient progress.

Gang Scheduling provides an environment similar to a dedicated machine, in which all the jobs progress together, and, at the same time, allows resources to be shared. In particular, preemption is used to improve performance in face of unknown runtime. This prevents short jobs from being starved in the queue waiting for long ones, and improves fairness. One problem with Gang Scheduling is that the requirement that all the jobs always run together causes too much fragmentation. This has led to several proposals for more flexible variants (53; 58). In conventional

Gang Scheduling, resources running tasks that perform I/O remain idle for the duration of the I/O operation. In paired Gang Scheduling (90) jobs with complementary characteristics are paired together, so that when a job performs I/O, the other can exec. Given a good job set, this can lead to improved resource utilization at little penalty to individual jobs.

A more general approach is to monitor the communication behavior of all applications, and try to determine whether they really benefit for Gang Scheduling. Gang Scheduling is then used for those that need it. Resources belonging to other jobs are used as filler to reduce the fragmentation cause by the scheduled jobs. Dealing with memory pressure, evaluations of Gang Scheduling assumed that all arriving jobs can be started immediately. Under high loads this could lead to situations where dozens of jobs share each resources. This is unrealistic as all these jobs would need to be memory resident or else suffer from paging, which would interfere with the synchronization among the jobs. A simple approach for avoiding this problem is to use admission controls, and only allows additional jobs to start if enough memory is available. While this avoids the need to estimate how much memory a new job will need, it is more vulnerable to situations in which memory becomes overcommitted causing excessive paging. When admission controls are used, and jobs wait in the queue, the question of queue order presents itself. The simplest option is to use a FCFS order. Improved performance is obtained by using backfilling, and allowing small jobs to move ahead in the queue. Using backfilling fully compensates for the loss of performance due to the limited number of jobs that are actually run concurrently. All the above schemes may suffer from situations in which long jobs are allocated resources, while short jobs remain in the queue waiting for execution. The solution is to use a preemptive long-range scheduling scheme. With this construction, the long term scheduler allocates memory to waiting jobs, and then the short term scheduler decides which jobs will actually run out of those that are memory resident. The long term scheduler may decide to swap out a job that has been in memory for a long time, to make room for a queued job that has been waiting for a long time.

2.1.5 Shop Scheduling

The *Open job shop scheduling problem* is a generalization of the *bipartite graph edge coloring problem* (66). A graph consist of a set of vertices *V* and a set of edges *E*. A graph as said to be *bipartite* when the set of vertices can be partitioned into two sets, *A* and *B*, such that every edge is incident to a vertex in the set *A*, and to a vertex in the set *B*. The bipartite graph edge coloring problem consists of assigning a color to each edge in the graph, in such a way that no two edges incident upon the same vertex are assigned the same color, and the total number of different colors utilized is the least as possible.

The open job shop scheduling problem consists of *m* machines, denoted by $M_1, M_2, ..., M_m$, which perform different tasks. There are *n* jobs $(J_1, J_2, ..., J_n)$, each of which consists of *m* tasks. The j^{th} task, of the job J_i, is denoted by the $T_{i,j}$, and it must be processed by the machine M_j for $p_{i,j} \geq 0$ time units. The total processing time for the job J_i is $p_i = \sum_j p_{i,j}$, the processing requirement for the machine M_j is $m_j = \sum_i p_{i,j}$, and *h* = $max\{p_i, m_j\}$ is defined as the finish time of every open shop schedule at least. The scheduling restrictions in an open shop problem are as follows:

1. Each machine can process at most one task at time.
2. Each job may be processed by at most one machine at time.
3. Each task $T_{i,j}$ must be processed for $p_{i,j}$ time units by machine M_j.

The open shop problem in which all the $p_{i,j}$ values are 0 or 1 is called the *unit-processing time* open shop problem. This open shop problem with the objective function of minimizing the makespan (i.e. the completion time of the last job of a given jobs set) corresponds to bipartite graph edge coloring problem. This results mapping the set of vertices *A* to the set of *jobs*, and the set of vertices *B* to the set of *machines*. An edge from a vertex in set A to a vertex in set B represents a task with unit processing time. Each color represents a time unit. The coloring rules guarantee that an edge coloring for the graph corresponds to the unit-processing time open shop problem. The makespan corresponds to the number of different colors used to color the bipartite graph.

A generalization of the open shop scheduling problem is the course timetable problem, in which professors can be viewed as machines, jobs can be viewed as classes, and the objective is to find times at which professors can instruct their classes, without any professor teaching more than one class at a time, and any class meeting with more than one professor at a time. In addition, the classical course timetable problem includes constraints where professors or classes cannot meet during certain time periods.

There are some important variants of the open shop scheduling problem. One of them is the *flow shop scheduling problem* (16; 68) that is defined as follow: a number of jobs are to be processed on a number of machines. Each job must go through all the machines in exactly the same order, and the job order is the same on every machine. Each machine can process at most one job at any time, and each job may be processed on at most one machine at any time. The objective is to find a schedule that minimizes the completion time of the last job. The main difference between the flow shop and the open job shop is that in the former problem the tasks for each job need to be processed in order, i.e., one may not begin processing task $T_{i,j}$ until $T_{i,j-1}$ has been completed for $1 < j \leq m$. In the open job shop problem the order in which tasks are processed is not important. In the open job shop problem jobs may have any number of tasks, rather than just m. Each task of each job is assigned to one of the machines rather than assigning the j^{th} task to machine M_j as in the flow shop problem. But, the order in which tasks must be processed in the open job shop problem is sequential as in the flow shop problem.

Another important variant of the open shop problem is the job shop problem. While in open job shop problem the different tasks of a job may be scheduled in any order, and in flow shop the order of the tasks is fixed and it is the same for all jobs, in the job shop problem the order of the tasks is fixed but it could be different for each job (87).

The last variant of the open job shop that we analyze is called *cycle shop*. The cycle shop can be defined as the special case of a job shop and the extension of a flow shop where all jobs have the same sequence of operations on the machines, but in contrast to a flow shop, some operations

can be repeated on some machines a number of times, and this number can differ from one machine to another (66).

2.1.6 Genetic Algorithms

Genetic Algorithms (GA) is a searching strategy in which an initial number of "guesses" is made to get an optimal solution (i.e. the initial population). Each guess is evaluated and assigned a "goodness" value (i.e. the fitness function). Those guesses with good values are selected, and new guesses are made by combining the existing guesses in a particular fashion (i.e. crossover). The guesses are evolved to the next generation on a survival of the fittest basis (i.e. selecting), thereby the "good" guesses are forwarded to the next generation, and the "bad" guesses are not. Random mutation is done to prevent the GA from getting struck in a local maximum. The general procedure of GA search is defined as follows:

1. Population generation: A population is a set of chromosomes each representing a possible solution, which is a mapping sequence between jobs and machines. The initial population can be generated randomly, or by using heuristics algorithms.

2. Chromosome evaluation: Each chromosome is associated with a fitness value, which is the makespan of the jobs-machine mapping this chromosome represents. The goal of GA search is to find the chromosome with optimal fitness value.

3. Crossover and Mutation operation: Crossover operation selects a random pair of chromosomes, and chooses a random point in the first chromosome. For the sections of both chromosomes, from that point to the end of each chromosome, crossover exchanges machine assignments between corresponding jobs. Mutation randomly selects a chromosome, then randomly selects a job within the chromosome, and randomly reassigns it to a new machine.

4. Finally, the chromosomes from this modified population are evaluated again. This completes one iteration of the GA. The GA stops

when a predefined number of evolutions is reached or all chromosomes converge to the same mapping.

GAs have been used in static scheduling (52; 54), where the schedule is generated before run-time, and dynamic scheduling (77; 94; 95), where the schedule is generated at run-time based on the run-time characteristics which are not possibly known beforehand.

Current Dynamic GA schedulers show near optimum solutions in simulations (77; 94; 95). In general, Dynamic GA schedulers are implemented in sequential algorithms, which take a long time to converge (65), but parallel versions of GA have been proposed to improve the algorithm performances (77).

2.1.7 Simulated Annealing

Simulated Annealing (SA) is a search technique based on the physical process of annealing, which is the thermal process of obtaining low-energy crystalline states of a solid (32). At the beginning, the temperature is increased to melt the solid. If the temperature is slowly decreased, particles of the melted solid arrange themselves locally, in a stable "ground" state of a solid. SA theory states that if temperature is lowered sufficiently slowly, the solid will reach thermal equilibrium, which is an optimal state. By analogy, the thermal equilibrium is an optimal job-machine association, the temperature is the total completion time of a schedule, and the change of temperature is the process of schedule change. If the next temperature is higher, which means a worse schedule, the next state is probability accepted. This because of the acceptance of some "worse" states provides a way to avoid local optimality, which occurs often in local search (32).

A SA algorithm is presented in (17). The initial system temperature is the makespan of the initial schedule (randomly generated). The first schedule is generated from a uniform random distribution. The schedule is mutated in the same way as for genetic algorithms, and the new makespan is evaluated. If the new makespan is better, the new schedule replaces the old one. If the new makespan is worse, a uniform random

number $z \in [0, 1)$ is generated. Then, z is compared with:

$$y = \frac{1}{1 + e^{(\frac{oldmakespan - newmakespan}{temperature})}}$$

If $z > y$, the new (poorer) schedule is accepted; otherwise it is rejected, and the old schedule is kept. So, as the system temperature "cools", it is more difficult for poorer solutions to be accepted. This is reasonable because when the temperature is lower, there is less possibility to find a better solution starting from another poorer one. After each mutation, the system temperature is reduced. This completes one iteration of SA. The heuristic stops when there is no change in the makespan for a number of iterations, or when the system temperature approaches zero.

2.2 Scheduling Frameworks

2.2.1 Multi-Site Scheduling Model

Multi-site computing is the execution of a job in parallel at different sites. This results in a larger number of totally available resources for a single job. In (34), Ernemann et al. address the potential benefit of sharing jobs between independent sites in a grid computing environment, and the aspect of parallel multi-site job execution on different sites. To this end, they propose various scheduling algorithms for several machine configurations.

The model proposed in (34) assumes a computing grid consisting of independent computing sites with their local workloads. This means that each site has its own computing resources as well as local users that submit jobs to the local job scheduling system.

In a typical single site scenario, all jobs are only executed on local resources. The sites may combine their resources and share incoming job submissions in a grid computing environment. Here, jobs can be executed on local and remote machines. The computing resources are expected to

be completely committed to grid usage. That is job submissions of all sites are redirected and distributed by a grid scheduler. This scheduler exclusively controls all grid resources.

In this scenario, the task of scheduling is delegated to a grid scheduler. The local scheduler is only responsible for starting the jobs after allocation by the grid scheduler. Three scenarios have been examined by Ernemann:

- Local Job Processing. This scenario refers to the common situation where the local computing resources at a site are dedicated only to its local users. A local workload is generated at each site. This workload is not shared with other sites.

- Job Sharing. In this scenario all jobs submitted at any site are delegated to the grid scheduler. The scheduling algorithms in grid computing consist of two steps: in the first step the machine is selected, the machine on which the job leaves the least number of free resources if started, and in the second step the allocation in time for this machine takes place using the Backfilling algorithm.

- Multi-Site Computing. This scenario is similar to job sharing: a grid scheduler receives all submitted jobs. Additionally, jobs can now be executed crossing site boundaries (see Figure 1). The adopted strategy for multi-site scheduling is based on a scheduler which first tries to find a site that has enough free resources for starting the job. If such a machine is not available, the scheduler tries to allocate the jobs on resources from different sites. To this end, the sites are sorted in descending order of free resources and allocating the free resources in this order for a multi-site job. In this case the number of combined sites is minimized. If there are not enough resources free for a job, it is queued and normal backfilling is applied.

Spawning job parts over different sites usually produces an additional overhead. This overhead is due to the communication over slow networks (e.g. a WAN). Consequently, the overall execution time of the job will increase depending on the communication pattern. For jobs with limited communication demand there is only a small impact. Note,

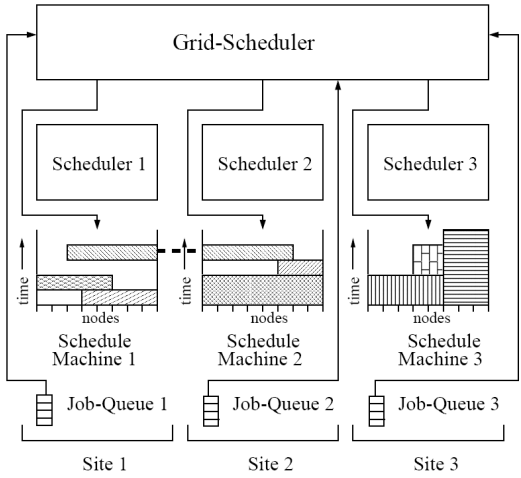

Figure 1: Support for multi-site execution of jobs.

without any penalty for multi-site execution, the grid would behave like a single large computer. In (34), they examine also the effect of multi-site processing on the schedule quality under the influence of a communication overhead.

The multi-site model designs a hierarchical architecture in which a high level scheduler assigns jobs to the lower level resources. In this model the local schedulers have to handle the job execution without performing any job scheduling decisions. The main difference between our approach and the multi-site model is due to the possibility of the high level scheduler, in multi-site model, to allocate a parallel job on resources belonging to different sites. This implies that the grid scheduler has a deep knowledge about the grid environment it operates. Whereas, one of our goals is to design a lightweight Meta-Scheduler exploitable in grid systems in which different providers share resources defining their own use policies. To this end, the local scheduler we propose is able to make scheduling decision and to influence the job execution at local level ac-

cording to the system administrator policies.

2.2.2 YML Framework

YML is a framework providing tools to parallelize applications which has been developed at CNRS/PRiSM in collaboration with InriaFuturs/LIFL (31; 75). It focuses on two major aspects: the development of parallel applications and their execution in a grid environment.

YML makes application development independent of the grid middleware used underneath and hides differences between them. On the YML point of view, an application is divided into different computing sections, each of them containing some tasks executed sequentially or concurrently. A task, called a component, is a piece of work that can be mapped to one node in a parallel environment. It has some input and output parameters and is generally reusable in different parts of the application as well as in different applications. YML provides a special type of components, called graph component, which consists of the description of sub graphs. YML divides the development of a parallel application in three major steps:

- A definition of new components. This definition consists in an Abstract and Implementation component description.
- A description of the parallel application. This description is independent of any underlying middleware and makes use of the components as functional units. It specifies parallel and sequential parts of the application using the YvetteML graph description language and provides notifications to synchronize the execution of dependent components. This description is directly deduced from the graph representation of the application.
- The compilation of the application. This step analyzes and transforms the application graph into a list of parallel tasks with respect of the precedence constraints.

The three steps are all middleware independent and ensure that no grid relevant knowledge is needed to develop parallel applications.

After the compilation of the application, the execution can be started using the Workflow Scheduler which will interact with the underlying middleware and submit tasks through the dedicated backend. Each task is launched by a YML worker which will contact the Data Repository Server to obtain the component binary and input parameters to start the computation. The use of Data Repository Servers hides the data migrations to the developer and ensures that the necessary data is always available to all components of the application.

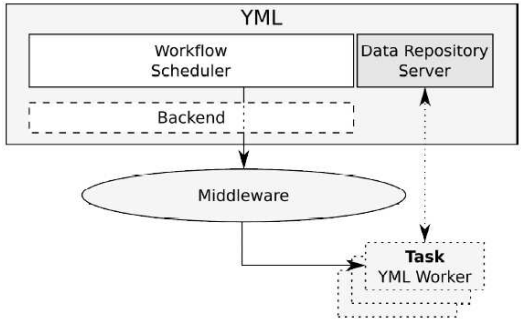

Figure 2: YML Workflow Scheduler interaction with the middleware.

The Workflow Scheduler (Figure 2) is currently adapted to propose multilevel scheduling features by integrating a new model based on an economic approach of resources. This model defines different entities, which will interact in the grid infrastructure. An entity can be a resource provider or consumer or both. Consumers require resources of the grid which are owned by providers. When a provider receives a request from a consumer, he will answer by proposing a set of suitable schedules and associated cost for parts of the application regarding access policy of the consumer and availability of local resources. He can possibly subcontract parts or the whole application to other resource providers without mentioning anything to the consumer. This model can be used in different scenarios: either cooperation or competition between sites in the grid infrastructure. Moreover, a hierarchy with different layers of scheduling instances can be built with this model. A new joining instance in the grid

infrastructure has to negotiate access policies with one or more resource providers. Each access policy will provide a set of static and dynamic parameters which will determine the usage conditions of the provided resources. These parameters will be used when an application is submitted, to obtain the list of suitable resources: static parameters represent permanent access conditions and do not require any interaction with other scheduling instances. In opposition, dynamic parameters will need to request each resource providers in order to filter the set of suitable resources. As presented in Figure 3, the main idea is to provide a YML server to each Local Resource Manager (LRM). This YML server has 3 main objectives: - to communicate with other YML servers and therefore, to connect the different clusters in a common grid; - to interact with the underlying LRM using a specialized backend; - to provide needed features missing in the LRM (as cluster (b) in Figure 3).

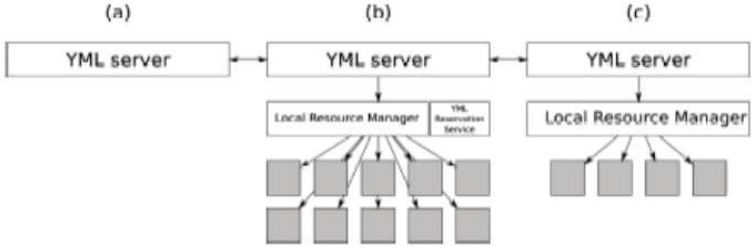

Figure 3: YML server cooperation.

When a user wants to execute an application on the grid infrastructure, he contacts the local YML server which analyzes the application and decides whether it can provide needed resources or not (eventually, it may have no local resources like cluster ((a) in Figure 3). It may forward the whole or parts of the request to other resource providers. This step will ensure collaboration between sites and a distribution of the global work in the whole infrastructure. Then, suitable schedules are sent back in return of each request and the user local YML server finally gathers the information and proposes different bids.

The most important difference between our approach to the scheduling problem and the YML approach, concerns with a central high level scheduler not present in YML framework. The Meta-Scheduler we propose is able to orchestrate the low level job assignment maximizing a given optimal criteria. The YML framework define different entities that cooperate to execute jobs. Moreover, YML is based on an economic model that drives the interaction among entities. Instead, our framework is based on a set of functions used to schedule jobs to low level scheduler, which implements scheduling algorithms to handle the job execution on specific resources. Another important difference between our approach and YML is the ability to classify incoming jobs with respect to their relevance. YML does not implement any mechanism to distinguish the importance of submitted jobs.

2.2.3 OAR Batch Scheduler

OAR has been designed as an open platform for research and experiments (23). The main contribution of OAR is its design, based on two high-level components: a SQL database (a central relational database engine), and the executive part.

The database is used to exchange information between modules, thus ensuring a complete opening of the system. The database engine is used to match resources (using SQL queries), and to store and exploit logging and accounting information. Additionally, a general purpose database can ensure friendly and powerful data analysis and extraction. The most important benefit of this approach is that the SQL language can be used for data analysis and extraction as well as for internal system management. Another advantage of using a standard database engine is that the scheduler should benefit of its robustness and efficiency. Although making SQL queries might induce some overhead compared to a specific implementation, the OAR engine has good behavior under high workload. The database engine does not represent a bottleneck for system scalability as it can handle efficiently thousands of queries simultaneously. Furthermore, robustness only depends on modules that have to let the system in

a coherent state and might otherwise fail without much harm.

The executive part is implemented using Perl, which is suited for system tasks. It is made of several modules, including one for launching and controlling the execution of jobs, and another for scheduling jobs. The monitoring tasks are handled by a separate tool that is invoked from OAR, and interfaced with the database. One of the goals of OAR is to make a research platform suited for scheduling experiments and resulting analysis. To help developers modifying the system, they made it modular and the implementation in a high-level language makes the system rather small and extensible. The submission of jobs in OAR works like in PBS (3): the interface is made of independent commands for submission, cancelation, or the monitoring. These commands are separated from the rest of the system, they send or retrieve information using directly the database and they interact with OAR modules by sending notifications to the central module. Job submission starts by a connection to the database to get the appropriate admission rules. These rules are used to set the value of job parameters that are not provided by the user, and to check the validity of the job submission. Possible parameters include a queue name, a limit on the job execution time, the number of needed nodes by the job. The rules are stored in the database and might be used to call an intermediate program so the admission can be elaborate.

OAR implements different functionalities such as: priorities on jobs, reservations, resources matching and backfilling scheduler.

The priorities are managed through submission queues. All the jobs are submitted to a particular queue which has its own admission rules, scheduling policy and priority. Reservations are a special case in which the user asks for a specific time slot. In this case, as long as the job meets the admission rules and the resources are available during the requested time slot, the schedule date of the job is definitively set. In OAR, resources required by jobs are matched with available ones as a user might need nodes with special properties (like single switch interconnection, or a mandatory quantity of RAM).

OAR uses backfilling and handles Best Effort jobs (jobs that can be canceled before the end of their allowed time). The scheduling of all the jobs in the system is computed by a module called metascheduler, which manages reservations and schedules each queue using its own scheduler. This module maintains an internal representation of the available resources similar to a Gantt diagram, and updates this diagram by removing time slots already reserved. Initially, the only occupied time slots are the ones on which some job is executing, and the ones that have been reserved. The whole algorithm schedules each queue in turn by decreasing priority using its associated scheduler.

The main difference between our scheduling framework and OAR concerns with the use of the framework. While our scheduler is designed to be exploited in a real grid environment, OAR is used for research and experiments. The use of a data repository, storing the information about jobs and users, can be exploited in our framework as a future work. We have not knowledge about executed jobs, or user peculiarities, except for those expressed by the job parameters. Moreover, OAR has a classification mechanism based on submission queues. Each queue has its own scheduling policy, and the order among jobs, stored in the same queue, is based on their arrival time. In the framework we propose, there is a single submission queue, each user submit its job to the Meta-Scheduler, which labels the jobs with a priority value computed exploiting a set of heuristics. This allows to manage different job constraints (e.g. increasing the number of heuristics), and to exploit reconfiguration mechanisms to adapt the classifier behavior (30).

2.2.4 KOALA Grid Scheduler

The execution of parallel jobs in grids may require co-allocation, i.e. the simultaneous allocation of resources such as processors and input files in multiple clusters. While such jobs may have reduced runtimes, because they have access to more resources, waiting for processors in multiple clusters, and for the input files, to become available in the right loca-

tions, may introduce inefficiencies. Moreover, as single jobs have to rely on multiple resource managers, co-allocation introduces reliability problems. In this section, we describe the design and implementation of a co-allocating grid scheduler named KOALA (74), which was developed at Delft University of Technology as a two-level scheduling strategy in grids, and which has been in operation in the Distributed ASCI Supercomputer (DAS) in the Netherlands since September 2005 (1).

KOALA defines jobs as a parallel application requiring files and processors that can be split up into several job components, which can be scheduled to execute on multiple execution sites simultaneously (co-allocation) (11; 26; 73). This allows the execution of large parallel applications requiring more processors than available on a single site (73; 74). Jobs that require co-allocation in grids may or may not specify the number and sizes of their components. Based on this, KOALA considers three cases for the structure of the job requests:

- Fixed request: The job request specifies the numbers of processors it needs in all the clusters from which processors must be allocated for its components.

- Non-fixed request: The job request specifies the numbers of processors it needs in the separate clusters, allowing the scheduler to choose the execution sites. The scheduler can either place the job components on the same or on different sites depending on the availability of processors.

- Flexible request: The job request specifies the total number of processors it requires. It is left to the scheduler to decide on the number of components, the number of processors for each component, and the execution sites for the components.

The KOALA scheduler consists of the following five components: the Co-allocator (CO), the Information Service (IS), the Data Manager (DM), the Processor Claimer (PC), and the Runners. The structure of KOALA is depicted in Figure 4.

CO is responsible for placing jobs, i.e. for finding the execution sites with enough idle processors for their components. CO chooses jobs to place, based on their priorities from one of the KOALA placement queues. If the components require input files, CO also selects the file sites for the components such that the estimated file transfer times to the execution sites are minimal. To decide on the execution sites, and file sites for the job components, CO uses one of the defined placement policies. Finding execution sites for the job components is done for non-fixed job requests.

IS is comprised of the Globus Toolkits Metacomputing Directory Service (MDS) (2; 41), and Replica Location Service (RLS), and Iperf, a tool to measure network bandwidth. A repository containing the bandwidths measured with Iperf is maintained and updated periodically. The MDS provides on request the information about the numbers of processors currently used, and the RLS provides the mapping information from the logical names of files to their physical locations. Requests to the MDS, and to the bandwidth repository, impose delays on placing jobs. Therefore, IS caches information obtained from the MDS, and the bandwidth repository, with a fixed cache expiry time. Furthermore, IS can be configured to do periodic cache updates from frequently used clusters before their cache expiry time.

DM is used to manage file transfers, for which it uses both Globus GridFTP (89) and Globus Global Access to Secondary Storage (GASS) (2). DM is responsible for ensuring that input files arrive at their destinations before the job starts to run.

After a job has been placed, it is the task of PC to ensure that processors will still be available when the job starts to run. If processor reservation is supported by local resource managers, PC can reserve processors immediately after the placement of the components. Otherwise, PC uses KOALAs claiming policy to postpone claiming of processors to a time close to the estimated job start time.

Runners are used to submit job components to their respective execution sites. They allow to extend the support for different application models in KOALA.

A job submitted to KOALA goes through four phases, which are plac-

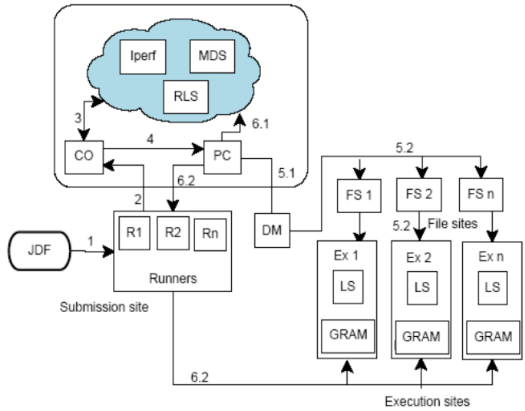

Figure 4: The interaction between the KOALA components.

ing its components, transferring its input files, claiming processors for its components, and starting and monitoring its execution (Figure 5).

In the first phase, a new job request arrives at one of the Runners (arrow 1 in Figure 4) in the form of a Job Description File (JDF). JDFs are specified using the Globus Resource Specification Language (RSL) (2; 41), with the RSL "+,-" constructs to aggregate the components requests into a single multi-request. After authenticating the user, the Runner submits the JDF to the CO (arrow 2), which in turn will append the job to the tail of one of the KOALA placement queues. CO then retrieves the job from this queue and tries to place the job components based on information obtained from IS (arrow 3). If the job placement fails, the job is returned to its respective placement queue. The placement procedure will be tried for the jobs in the placement queues at fixed intervals for a fixed number of times.

Phase 2 starts by CO forwarding the successfully placed job to PC (arrow 4). On receipt of the job, PC instructs DM (arrow 5.1) to initiate the third-party file transfers from the file sites to the execution sites of the job components (arrows 5.2).

In phase 3, PC estimates the job start time, and the appropriate time

that the processors required by a job can be claimed. At this time, and if processor reservation is not supported by the local resource managers, PC uses a claiming policy to determine the components that can be started based on the information from IS (arrow 6.1). It is possible at the job claiming time for processors not to be available anymore, e.g. they can then be in use by local jobs. If this occurs, the claiming procedure fails, the job is put into the claiming queue, and the claiming is tried again at a later time.

In phase 4, the Runner, used to submit the job for scheduling in phase 1, receives the list of components that can be started (arrow 6.2), and forwards those components to their respective execution sites. At the execution sites, the job components are received by the Globus Resource Allocation Manager (GRAM), which is responsible for authenticating the owner of the job, and sending the job to the local resource manager for execution.

KOALA exploits a queue-based job classification mechanism: jobs are stored in different queues according to their relevance, and each queue has its own scheduling policy. Another important difference, between KOALA and our scheduling framework, is that KOALA needs the Globus middleware support to be used. Globus implements a lot of functionalities used by KOALA to access, and to handle, the grid infrastructure. This could be an advantage, because of Globus ensures the use of the scheduler in many different contexts both scientific and industrial, but could also be a disadvantage, because of without Globus support KOALA can not be exploited.

2.2.5 Moab and Maui scheduler

Maui determines when and where submitted jobs should be run (4). Jobs are selected and started in such a way that is not only enforced a site's mission goals, but also improved the resource usage, and minimized the average job turnaround time.

Mission goals are expressed via a combination of policies which con-

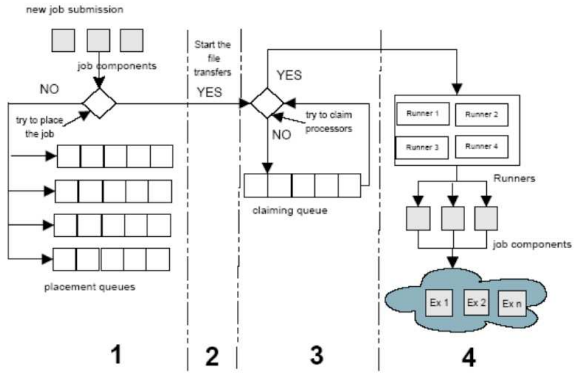

Figure 5: The four phases of job scheduling in KOALA.

strain how jobs will be started. A number of base concepts require review to set the groundwork for a detailed discussion of the algorithms. Maui schedules on a iterative basis scheduling, followed by a period of sleeping or processing external commands. Maui will start a new iteration when one or more of the following conditions is met:

- a job or resource state-change event occurs (i.e. job termination, node failure);
- a reservation boundary event occurs;
- the scheduler is instructed to resume scheduling via an external command;
- a configurable timer expires.

Maui supports the concept of job class, also known as job queue. Each class may have an associated set of constraints determining what types of jobs can be submitted to it. Constraints can also be set on a per-class basis, specifying which users, groups, etc., can submit to the class. Further, each class can optionally be set up to only be allowed access to a particular subset of nodes. Within Maui, all jobs are associated with a class.

Maui also supports the concept of Quality of Service levels. These QoS levels may be configured to allow many types of special privileges, including adjusted job priorities, improved queue time and expansion factor targets, access to additional resources, or exemptions from certain policies.

Each batch job submitted to Maui is associated with a number of key attributes or credentials describing job ownership. These credentials include the standard user and group ID of the submitting user. However, they also include an optional account, or project, ID that can be used in conjunction with allocation management systems.

Maui's scheduling behavior can be constrained by way of throttling policies, policies which limit the total quantity of resources available to a given credential at any given moment. The resources constrained include things such as processors, jobs, nodes, and memory.

The *Moab Workload Manager* is based on the Maui batch scheduler (93), with all its flexibility and added features: backfilling, service factors, resource constraints and weights, fair-share options, direct user/group/account prioritization, target wait times, etc. However, based on the Maui Scheduler Administrators Guide its default behavior out of the box is a simple FCFS batch scheduler, with a backfilling policy that maintains a time reservation for the first job in the queue, i.e. the EASY backfilling.

In (93), it was verified that, in the Maui scheduler, the priority of each job is computed as a weighted sum of several factors, where the weights are set by the system administrator. Each factor itself is a weighted sum of sub-factors, whose weights, again, are governed by the system administrator. The authors found that even though all the factor's weights are set to 1 (in an array called *CWeight*), all the weights of the subfactors are set to 0, except for that of the job's queue time which is set to 1 (all the sub-factors weights are saved in the *SWeight* array). The result is that the job's queue time is the only factor that is not zero, and even though the factor weights are set to 1, the queue time is the priority functionresulting in a FCFS scheduler.

As for the other analyzed frameworks, Maui exploits a job classification based on different queue scheduling strategies. Users can submit their jobs to specific queue (or class) with respect to some credentials. Furthermore, Maui could exploit different, simple, scheduling algorithms, but, due to its default configuration, Maui uses the FCFS algorithm since the system administrator does not change the scheduling policy.

2.2.6 Nimrod-G

Nimrod-G is a computational economy-based grid resource management and scheduling system, which supports deadline- and budget-constrained algorithms for scheduling parameter sweep (task and data parallel) applications on distributed resources (8; 19; 21). It provides a simple parameter specification language for creating parameter-sweep applications (i.e. application containing large number of independent jobs operating on different data sets). The domain experts can create a plan for parameter studies, and use the Nimrod-G broker to handle all the issues related to the seamless management and execution, including resource discovery, mapping jobs to appropriate resources, data and code staging, and gathering results from multiple grid nodes back to the home node (i.e. a node from which a job request originates). Depending on the user's requirements, Nimrod-G dynamically leases grid services at run-time based on their availability, capability, and cost.

Nimrod-G is able to address job's deadline constraints when it performs the jobs scheduling. Such a guarantee of service is hard to provide in a grid environment as its resources are shared, heterogeneous, distributed in nature, and owned by different organizations having their own policies and charging mechanisms. In addition, scheduling algorithms need to adapt to the changing load and resource availability conditions in the grid, in order to achieve performance, and at the same time meet cost constraints. In Nimrod-G application level resource broker (also called an application level scheduler) for grid, they have incorporated three adaptive algorithms for scheduling:

- Cost Optimization, within time and budget constraints,

- Time Optimization, within time and budget constraints,
- Conservative Time Optimization, within time and budget constraints.

The Time Optimization scheduling algorithm attempts to complete the experiment as quickly as possible, within the budget available. A description of the core of the algorithm is as follows: 1. for each resource, calculate the next completion time for an assigned job, taking into account previously assigned jobs and job consumption rate. 2. sort resources by next completion time. 3. assign one job to the first resource for which the cost per job is less than or equal to the remaining budget per job. 4. repeat the above steps until all jobs are assigned.

The Cost Optimization scheduling algorithm attempts to complete the experiment as economically as possible within the deadline. 1. sort resources by increasing cost. 2. for each resource in order, assign as many jobs as possible to the resource, without exceeding the deadline.

The Conservative Time Optimization scheduling algorithm attempts to complete the experiment within the deadline and cost constraints, minimizing the time when higher budget is available. It spends the budget cautiously and ensures that a minimum of "the budget-per-job" from the total budget is available for each unprocessed job. 1. split resources by whether cost per job is less than or equal to the budget per job. 2. for the cheaper resources, assign jobs in inverse proportion to the job completion time (e.g. a resource with completion time $= 5$ gets twice as many jobs as a resource with completion time $= 10$). 3. for the dearer resources, repeat all steps (with a recalculated budget per job) until all jobs are assigned.

Compared to all the other analyzed scheduling frameworks, Nimrod-G is the only one that addresses the job deadline constraint. The Nimrod-G perspective is different from our because it is based on an economic model, and it combines the concept of job deadline to the concept of bud-

get. This means that the scheduler has to assign jobs is such a way to respect their deadlines, but at the same time, it has to respect also a cost limit as another jobs constraint. Nimrod-G does not perform any job classification. It considers only the jobs QoS, expressed as deadline and budget parameters, without establishing a relation among them. Eventually, Nimrod-G dose not address the jobs software licence requirements.

2.2.7 AppLeS: Application Level Scheduling

Application Level Scheduling (AppLeS) (15) is a methodology for adaptive application scheduling on Grid systems. The goals of the AppLeS project have been twofold. The first goal concerns to investigate adaptive scheduling for grid systems. The second goal aims to apply research results to applications, for validating the efficacy of AppLeS's approach and, ultimately, extracting grid performance for the end-user. AppLeS's researchers have achieved these goals via an approach that incorporates static and dynamic resource information, performance predictions, application and user-specific information, and scheduling techniques that adapt application execution "on-the-fly". Based on the AppLeS methodology, they have developed template based grid software development, and execution systems, for collections of structurally similar classes of applications. Each application is fitted with a customized scheduling agent that monitors available resource performance and generates, dynamically, a schedule for the application.

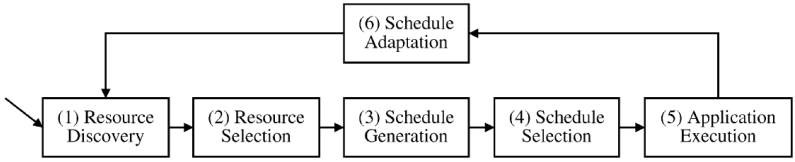

Figure 6: Steps in the AppLeS methodology.

The individual steps followed by an AppLeS agent are depicted in Figure 6, and are detailed below:

1. Resource Discovery. The AppLeS agent must discover the resources that are potentially useful to the application. This can be accomplished using a list of the users logins or by using ambient grid resource discovery services (59).

2. Resource Selection. The agent identifies and selects viable resource sets from the possible resource combinations. AppLeS agents typically use an application-specific resource selection model to develop an ordered list of resource sets (29). Resource evaluation typically employs performance predictions of dynamically changing system variables (e.g. network bandwidth, CPU load), and/or values gathered from previous application executions.

3. Schedule Generation. Given an ordered list of feasible resource sets, the AppLeS agent applies a performance model to determine a set of candidate schedules for the application on potential target resources. In particular, for each set of feasible resources, the agent uses a scheduling algorithm to determine the best schedule for the application on just the target set (i.e. for any given set of resources, many schedules may be possible).

4. Schedule Selection. Given a set of candidate schedules (and the target resource sets for which they have been developed), the agent chooses the best overall schedule that matches the users performance criteria (execution time, turnaround time, convergence, etc.).

5. Application Execution. The best schedule is deployed by the AppLeS agent on the target resources using whatever infrastructure is available. For some AppLeS, ambient services can be used (e.g., Globus (2; 41)). For other AppLeS applications, deployment may be performed on the bare resources by explicitly logging in, staging data, and starting processes on the target resources (e.g., via ssh).

6. Schedule Adaptation. The AppLeS agent can account for changes in resource availability by looping back to Step 1. Indeed, many grid resources exhibit dynamic performance characteristics, and resources may even join or leave the grid during the applications lifetime. AppLeS targeting long-running applications can then iteratively compute and implement refined schedules.

AppLeS agents are integrated pieces of software in which the application code and the agent are combined and not easily separated. In particular, it is difficult to adapt an AppLeS application to create a different AppLeS application. Moreover, application developers have to enhance their own application supporting the AppLeS code. In order to ease this programming burden, AppLeS templates have been developed. These templates embody common features from various similar (but not identical) AppLeS-enabled applications. An AppLeS template is a software framework developed so that an application component can be easily "inserted" in modular form into the template to form a new self-scheduling application. Each AppLeS template is developed to host a structurally similar class of applications.

The AppLeS approach is structured according to a decentralized architecture: each AppLeS-enabled application is integrated with its own agent, which provides scheduling and execution management mechanisms. In this way, each application is able to select the most profitable resource set to be executed, and it can choose the best schedule that respects the users performance requirements. Even if, from the users point of view, this approach can guarantee a good level of QoS requirements respect, it needs a big effort in the application development phase: the application has to integrate the AppLeS agent code. This is the main difference between AppLeS and our approach, our scheduling framework does not impact on the programmers works.

2.2.8 Portable Batch System

The Portable Batch System (PBS) comes in two flavors: OpenPBS (84), which is intended for small clusters, and PBS-Pro (10), which is the full fledged, industrial strength version (both are descendants of the system described in (51)). Here, we will focus on PBS-Pro.

Schedulers included with the PBS-Pro are FCFS, *Shortest Job First* (SJF), user/group priorities, and fair-share. Also, site specific schedulers can be implemented natively in the C and TCL programming languages, or in a special language called BaSL. Other features include checkpoint support, re-pack and rerun support for failed or stopped jobs, and failed nodes recovery. The fair-share scheduler uses a hierarchical approach. The system administrator can distribute a subset of resources among groups, whose resources can be divided into subgroups. This creates a tree structure in which each node is given resources, which are distributed by the system administrator assigned ratios to its child nodes, all the way down to the tree leaves. The leaves themselves can be either groups or specific users.

The system administrator can define work queues with various features. Queues can have resource limits that are enforced on the jobs they hold. A job can even be queued according to its specified resource requirements the system administrator can define a queue for short jobs, and the queuing mechanism can automatically direct a job with small CPU requirements to the short jobs queue. The system administrator can define a priority for each queue, setting the scheduling order among queues, or can be selected for schedule in a round robin fashion. Queues can also be set inactive for certain times.

The PBS-Pro system support preemption between different priority jobs. The system administrator can define a preemption order among queues, by which jobs from higher priority queues can preempt jobs from lower priority queues, if not enough resources are available. Inter-queue preemption is enabled by default, but there is only one default queue. The default scheduler in both PBS systems is SJF. To prevent starvation (i.e. which is the main problem of SJF scheduling), PBS-Pro can declare a job as starving after some time it has been queued (i.e. with the default

time set to 24 hours). A starving job has a special status no job will begin to run until it does. The result is that declaring a job as starving causes the system to enter a draining mode, in which it lets running jobs finish until enough resources are available to run the starving job. The starvation prevention mechanism can be enabled only for specific queues. Backfilling is supported, but only in context of scheduling jobs around a starving job waiting to run, and only if users specify a wall time CPU limit. Like the starvation prevention mechanism, backfilling can also be enabled for specific queues.

In general, the Portable Batch System is used as a centralized scheduler, which support job characterization using queues with different scheduling policies. PBS is not able to handle resources spread on different administrative domains. This is the main difference with the scheduler we propose, in which each job has its own characterization, and our Meta-Scheduler is able to schedule jobs on distributed resources managed by their own scheduler.

2.3 Conclusions

In this Chapter we analyzed the most important scheduling algorithms proposed in literature, and the most common scheduling frameworks.

We analyzed the most known scheduling algorithms, and we exploited the Backfilling strategy to propose a novel solution for the scheduling problem described in section 1.2. We proposed a variant of Flexible Backfilling, which is able to characterize incoming jobs according to their relevance (with respect to job deadlines, software licence requirements, etc.), and to schedule them according to the order inferred by their priorities. Moreover, we proposed a new scheduling algorithm: the Convergent Scheduler. Compared with the exiting algorithms, the Convergent Scheduler addresses the new aspects concerning the job deadlines, and the software licence requirements. Convergent Scheduler is also able to compute for each job a value that specifies the degree of preference for each machine in the computing platform.

For each analyzed framework, we pointed out the main differences with our approach, and we designed a path through the features not addressed by the exiting strategies, and covered by the scheduler we proposed.

Chapter 3

A scalable and distributed scheduling approach

In this chapter we present the core activities of our research. The Thesis topic concerns with the definition of a scalable and dynamic scheduler for batch jobs exploitable in grid environments. The objective of our work is to develop a scheduler able to manage the execution of a large number of jobs on a grid platform composed by thousands machines located in different sites. We suppose that a job is sequential or multi-threaded, that it is executed only on a single machine, that jobs are allocated to a machine according to the space sharing policy, and that jobs are independent (i.e. the execution of a job does not depend on the execution, or results, of previous jobs). Furthermore, our scheduler must be able to dispatch jobs according to the "Jobs arrive over time" on-line paradigm (87). This because users can submit their jobs to the system at any time without any predefined policy.

We design our scheduling framework according to the hierarchical model (18). The scheduler at the top of the hierarchy is called *Meta-Scheduler* (MS), and the one at the resource level it is called *Local-Scheduler* (LS). Figure 7 shows the two-level grid scheduler architecture we are working on. Our research focuses on two key points:

- to design a MS able to efficiently schedule incoming jobs balancing

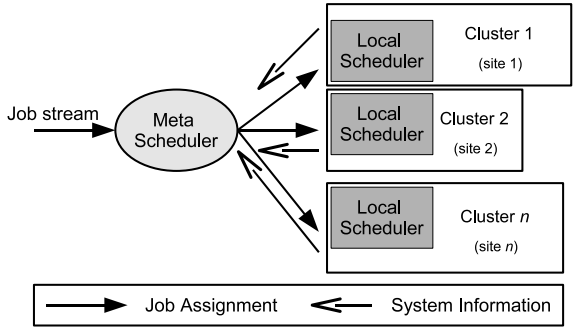

Figure 7: Two-levels scheduler architecture.

the workload among the underlying clusters,

- the study of a LS able to schedule jobs among the cluster's machines it manages, in order to maximize the hardware/software resource usage, and to maximize the number of jobs that are executed respecting their requested QoS (e.g. deadlines, software licence requirements).

Concerning the Meta-Scheduler (Section 3.1), our efforts are devoted to define new jobs classification mechanism, in order to improve the quality of the MS scheduling decisions, and to study new scheduling policies able to balancing the workload among clusters exploiting the carried out classification. The Meta-Scheduler is characterized by two phases as shown in Figure 8: a job classification phase, in which each job is labeled with a priority value, and a scheduling phase, in which jobs are dispatched among the Local-Schedulers.

The MS classifier exploits a set of heuristics each one managing a specific job parameter. These heuristics are based on static information describing the submitted jobs, without exploiting dynamic information about the computational environment (e.g. LS's job queues status, software licences availability, machines status, and jobs approaching their deadlines). In section 5.1, we studied how to apply autonomic features to our classifier in such a way that it is able to change its own behavior

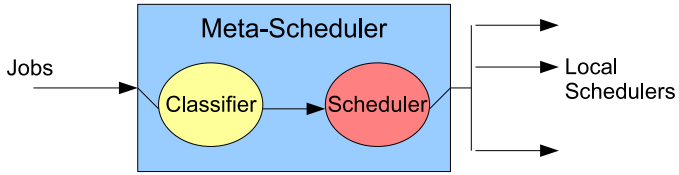

Figure 8: Meta-Scheduler architecture.

and adapt itself to the operating environment.

The MS's scheduling phase is based on two functions: *Load* and *Ordering*. The former is used to schedule jobs among clusters balancing the workload, the latter is used to balance the number of jobs with equal priority in each cluster queue. When the clusters workload is balanced, Ordering allows to spread jobs with the same priority to different clusters, in such a way that the highest priority jobs can be executed before lower priority ones.

Concerning the Local-Scheduler level (Section 3.2), we exploit two different solutions: a variant of the well known Flexible Backfilling algorithm (9), and a new approach called Convergent Scheduling (43; 83).

Our Flexible Backfilling (FB) assigns a priority value to each job in the submission queue considering both the optimization of resource usage, and a set of job QoS requirements (e.g. job deadlines, software licence requirements). Queued jobs are ordered according to their priority values: the job with the highest priority value is stored at the head of the queue. Job priorities are computed at each scheduling event (i.e. at a job submission and at a job ending), exploiting a set of heuristics, each one managing a specific job constraint. Our FB assigns to the first job in queue a resource reservation that is preserved until the job starts its execution.

The Convergent Scheduling (CS) technique is a novel scheduling algorithm for batch jobs, which allows to make an effective job scheduling plan considering at the same time: resources status, job QoS requirements, and administrator constraints. One of the innovative aspects of the CS is the ability to compute a job priority related to each available

machine. Priorities express the degree of preference among a (running or queued) job and the cluster's machines. Jobs are labeled with a priority value using a set of heuristics: *Deadline*, *Licences*, *Input*, *Wait minimization*, *Minimal requirements*, *Overhead minimization* and *Aging*. Each heuristics manages a specific constraint, and it increases/decreases the matching degree between a job and a machine. CS aims to schedule a subset of queued jobs that maximize the degree of preference for the available resources, and that can be simultaneously executed without violating the constraints on the licence usage.

3.1 Meta-Scheduler

3.1.1 Classification Phase

In the classification phase each submitted job is labeled with a priority value. We define a range for assignable priorities, i.e. each job priority belongs to $[1 - 10]$. Priorities are computed by using a set of heuristics each one analyzing a different job QoS requirement: job deadline, software licence requirements, submitting user. At this level, we define the priority as a value that "expresses and aggregates" the whole job QoS requirements. In this way, we can use it to easy compare jobs with different QoS requirements, and to infer a job execution order: high priority jobs will be executed before lower ones.

Priorities are assigned when jobs are submitted to the system, and are exploited in the scheduling phase to dispatch jobs among clusters (Section 3.1.2). The job priority is function of only job's parameters and it does not consider information about computational resources such as: number and type of cluster's machines, software licences availability, and machines workload. The goal of our classification phase is to exploit job attributes and characteristics to enable a classification in an independent way, with respect to the features of the computing platform used.

In our classifier, the job priority values are carried out by averaging the weighted contributions, $\triangle_{D_i}$, $\triangle_{L_i}$, and $\triangle_{U_i}$, computed for each job i

by three heuristics: *Deadline*, *Licences*, and *User*.

The Deadline heuristics is used to evaluate how much a job is "far" from the time at which it has to be done its execution. The contribution of Deadline ($\triangle_{D_i}$) to the priority value of a job i is proportional to the "proximity" of the time at which i must start its execution to meet its deadline. The Deadline heuristics is introduced in order to improve the number of jobs that execute respecting their deadline. Jobs closer to their deadline get a boost in preference that gives them an advantage in scheduling. The problem we have to solve to implement our heuristics is to give a meaning to the word "proximity": how can we estimate when a job is "proximity" to its deadline?

Our classifier has not knowledge on resources, and it cannot estimate if some available machines can perform the analyzed job within its deadline. The only information it can exploit are those describing the submitted jobs. The solution we find to this problem is based on the $Margin$ value computed for each job.

Let $Margin_i$ be the difference between the time at which i must start its execution, in order to respect its deadline, and the i submission time (Figure 9):

$$Margin_i = Deadline_i \ - \ Estimated_i \ - \ Submission_i$$

Where $Deadline_i$ is the deadline of the job i, $Estimated_i$ is its estimated execution time, and $Submission_i$ its the time at which i is submitted. To compute the $\triangle_{D_i}$, Deadline exploits the average margin value $E[Margin]$ computed as:

$$E[Margin] = \frac{\sum_{i=1}^{|N|} Margin_i}{N}. \tag{3.1}$$

Where N is the window size, which specifies the number of jobs that contribute to compute $E[Margin]$, and analyzed before i.

A job i is considered a "proximity" one if $Margin_i < E[Margin]$, otherwise it is considered to be a "faraway" one.

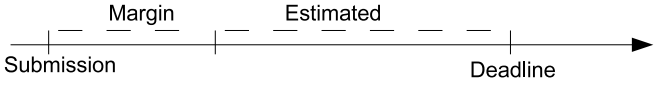

Figure 9: Graphical representation of the job parameters: Submission, Estimated, Margin and Deadline.

Figure 10: Graphical representation of the $[0, E[margin] * 2]$ interval subdivision.

To compute $\triangle_{D_i}$, the heuristics considers the double value of $E[Margin]$, and a job classification policy according to an exponential distribution, in which the number of jobs with priority P is exponentially greater than the number of jobs with priority $P - 1$. In order to improve the probability to meet the job deadline, we use the $E[Margin] * 2$ that permits to over-estimate the time interval in which a job is a "proximity" one, while the exponential priority distribution permits to limit the number of jobs to which the highest priority is assigned. This way allows to better satisfy the job QoS requirements, by exploiting a more fine granularity in the job priority process assignment.

Expression (3.2) formalizes the priority distribution we used to configure the Deadline heuristics. Let s and j be two integer representing two different priority values, with $s < j$, let $\#job_j$ be the number of jobs with priority value equal to j, then Deadline behaves at the best when:

$$\#job_j = \frac{\#job_s}{2^{j-s}}. \tag{3.2}$$

In order to assign the priority values to jobs using their $Margin$ value, the interval $[0, E[Margin] * 2]$ is divided into subintervals as depicted in Figure 10. The subintervals $Interval$ are computed as:

$$Interval_{(max-k)} = [S_k, S_{k+1}] \text{ with } k = 0, ..., max - 1$$

Where max is a value set by the system administrator, and representing the highest priority value that a job can assume. S_k is computed as:

$$\begin{cases} S_0 = 0 \\ S_k = S_{k-1} + MinUnity \cdot 2^k \end{cases} \tag{3.3}$$

Where $MinUnity$ is given by:

$$MinUnity = \frac{2 * E[Margin]}{\sum_{k=1}^{max} 2^k} \tag{3.4}$$

We fix $\triangle_{D_i}$ equal to $(max - k)$, i.e. equal to the index of $Interval$ to which the job $Margin$ value belongs to. If a job has $Margin$ greater than $E[Margin] * 2$, we consider the job a "faraway" one with good reason, and we assign to it a $\triangle_{D_i}$ equal to the lowest priority. In this way, all incoming jobs can be classified using the Deadline heuristics.

Exploiting the described procedure, it can happen that jobs with both large estimated execution time and large $Margin$ obtain lower priorities than jobs with small estimated execution time and small $Margin$. It can lead to execute large jobs without respecting their deadline. To face this aspect, we introduced the value R_i computed as:

$$R_i = \frac{Margin_i}{Estimated_i}$$

$R_i \geq 1$ means that the job i can be classified as a "faraway" one, and, in this case, $\triangle_{D_i}$ is updated as:

$$\begin{cases} \triangle_{D_i} = \triangle_{D_i} - \lfloor R_i \rfloor & \text{if } (\triangle_{D_i} - \lfloor R_i \rfloor) \geq 0 \\ \triangle_{D_i} = 0 & \text{if } (\triangle_{D_i} - \lfloor R_i \rfloor) < 0 \end{cases}$$

$R_i < 1$ means that the job i can be classified as a "proximity" one. So, let consider the interval $[0, 1]$ divided into subintervals Sub_w computed as:

$$Sub_w = [\frac{1}{2^{w+1}}, \frac{1}{2^w}] \text{ with } w = 0, ..., max - 1$$

and the index w of Sub_w to which R_i belongs to, $\triangle_{D_i}$ is updated as:

$$\begin{cases} \triangle_{D_i} = \triangle_{D_i} + w & \text{if } (\Delta_{D_i} + w) \leq max \\ \triangle_{D_i} = max & \text{if } (\triangle_{D_i} + w) > max \end{cases}$$

The Licence heuristics computes $\triangle_{L_i}$ to favor the execution of jobs that improve the contention on the software licences usage, i.e. the jobs that require a great number of different licences to be executed. $\triangle_{L_i}$ is computed as a function of the number of different software licences required by a job, in such a way that jobs asking for a high number of licences get a boost in preference that gives them an advantage in scheduling. This pushes jobs using many licences to be scheduled first, this way, releasing a number of licences.

Licence does not consider dynamic information about the number of copies available for each type of licences at any time. In this way, Licence is exploitable even if the underlying clusters have different sets of licences with a different number of available copies for each of them. Our heuristics takes into account only the different types of licences requested by each job, without considering their availability. In this way, it is independent from the dynamic status of the licence availability.

To compute $\triangle_{L_i}$ the heuristics considers the number $|L|$ of different types of licences. The interval $[0, |L|]$ is divided in max subintervals of the same size $|L|/max$. So, each subinterval corresponds to a priority value (e.g. at the first subinterval corresponds to the lowest priority value). $\triangle_{L_i}$ is computed as a function of the number of licences request by a job, and it is fixed equal to the related subinterval index. For instance, let the number of different licences $|L| = 20$, $max = 10$ and a job i requesting for 5 different types of licences to be executed. The $\triangle_{L_i}$ computed by Licence is 3, because of the 5 licences requested by i belongs to the third subinterval $[4, 6)$ in $[0, |L|]$ (Figure 11).

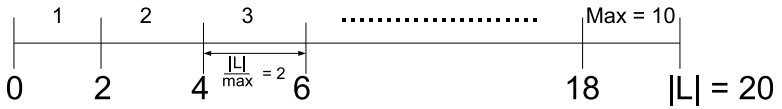

Figure 11: Licence: example of interval subdivision.

The User heuristics computes $\triangle_{U_i}$ in order to execute jobs respecting the user peculiarities. We defined three classes of users: *Gold*, *Silver*, and *Regular*, to which are assigned decreasing priority values. Such priority values can be function of several parameters, which are defined by the agreement between each user and system provider, such as: user importance, kind of requested resources/services, and kind of project. We suppose that each user can label its jobs with a priority value called *owner priority* in the range $[0, 3]$. In this way the user can specify an importance for each of its job. We assign a predefined priority value to each class of user, then we add the owner priority to this value to carry out the $\triangle_{U_i}$.

3.1.2 Scheduling Phase

The scheduling phase aims to dispatch jobs balancing the workload among the underlying clusters. We design the MS's scheduling phase following two objectives:

- the information exchanged with the underlying level have to be the least as possible,
- MS has to be able to exploit the job priorities in order to improve the quality of its scheduling decisions.

The first objective drives us when designing a lightweight Meta-Scheduler easy to interface with different scheduling algorithms exploited at Local-Scheduler level. We study the scheduling phase in order to bind the interaction between MS and LS to few types of messages: jobs

to be executed are sent from MS to LS, and information about already executed jobs are sent from LSs to MS.

We design the Meta-Scheduler in such a way that it is able to estimate the cluster's workload considering the jobs stored in each LS's queue (i.e. the workload due to past assignments to each LS). Using these information, MS selects the cluster to send jobs balancing the workload. In this way, we avoid the generation of high communication between levels.

What is relevant in this interaction protocol is that no dynamic information about software/hardware resources are exchanged among levels, but the interaction involves only the jobs that are sent from MS to LSs, and information about executed jobs that are sent from LSs to MS. In this way, our MS is able to interact with schedulers involved in different domains, such that no local policies are violated or will be changed. This choice permits to exploit our MS in grid environments, in which each system administrator has its own resource management policies, and does not agree to share information regarding its resources with other systems.

The second objective of the scheduling phase aims to improve the quality of the schedule when MS identifies a subset of clusters with approximately the same workload. This because of MS uses an estimation of each cluster workload to provide the balance among them. Also in this case, MS is able to schedule jobs preserving the workload balance but, randomly choosing a cluster in this subset, it can happen that a number of jobs with the same priority could be assigned to only few clusters. This could lead some jobs to be executed violating their QoS requirements.

To avoid this phenomenon, we enrich our MS with a scheduling function that aims to distribute jobs with the same priority to different LSs. This implies that jobs with critical QoS requirements (high priority) are spread on different clusters, are executed before low priority ones (i.e. jobs with weak QoS requirements), and do not conflict each other. In this way the number of executed jobs respecting their QoS requirements is improved.

To reach the described objectives, we design the scheduling phase according to a policy based on two functions: *Load* and *Ordering*. Load aims to balance the workload among clusters by assigning a job to the less loaded cluster (i.e. the first objective). The workload on a cluster is estimated summing the load due to the jobs queued to it. Ordering aims to balance the number of jobs with equal priority in each cluster queue (i.e. the second objective).

According to Load, clusters are ranked, and a job is scheduled to the cluster with the smallest rank. It works according to the following principle:

The best cluster to assign a job is the idle one or the one with the minimum load, due to the jobs with priority equal to or greater than the priority of the job currently analyzed.

To estimate the workload on each cluster, an array of max positions (the number of possible priority values that MS can assign to submitted jobs) is defined for each one. Each priority value corresponds to an array position, which stores the amount of workload due to jobs with the corresponding priority value, plus the amount of the load due to jobs with higher priority. Accordingly, the first array position (when the array elements are arranged in increasing order) stores the workload due to all the jobs queued to a LS.

An example of the data structure used by the Load function is shown in Figure 12. Here, the value in the fourth position indicates that the load due to the jobs with priority 4, plus the load due to jobs with priorities higher than 4 is 25. In particular, analyzing two consecutive array positions, for instance 4 and 5, we can infer that the load due to jobs with priority 4 is 7, and the load due to jobs with priority higher than 4 is 18. Moreover, there are not jobs with priority 1, 2, and 3, and the total workload due to the queued jobs is 25.

When a job is assignable to some eligible clusters, the problem is to find the cluster that can run it as soon as possible, by improving the number of jobs that are executed respecting their QoS requirements.

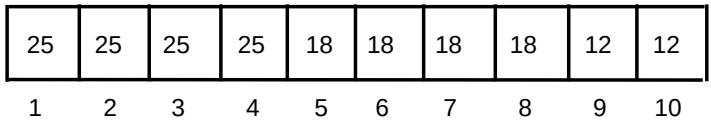

Figure 12: Example of array storing the workload due to the jobs queued at LS level.

Supposing to have a job i with priority P. The Load function uses P to access the cluster arrays. The first found cluster with 0 in the $P\text{-}th$ array position is the chosen one for the job assignment. Indeed, according to the defined principle, it is considered an idle cluster and it is the one that potentially can quickly start the execution of the job. This approach is also valid when there are some queued jobs that have lower priority than the analyzed one. When the value stored in the $P\text{-}th$ position of each array is greater than 0, the job is queued to the LS whose array stores in such position the lowest value. It will be selected to be scheduled later, when the execution of higher priority jobs, and jobs with the same priority, but early arrived, will be completed.

To balance the number of the jobs with equal priority in each cluster queue, we introduce the Ordering function. It works according to the following principle:

The best cluster to assign a job is the one with the minimum number of queued jobs, with priority equal to the priority of the job currently analyzed.

Ordering exploits a technique analogous to that used by the Load function. It uses an array of max positions for each cluster.

Each priority value corresponds to an array position, which stores the number of queued jobs with the corresponding priority value, plus the number of queued jobs with higher priority than the one specified by the array position.

An example of the data structure used by the Ordering function is shown in Figure 13. Here, the value 9 in the fourth position indicates that there are 3 jobs with priority 4 and 6 with higher priority.

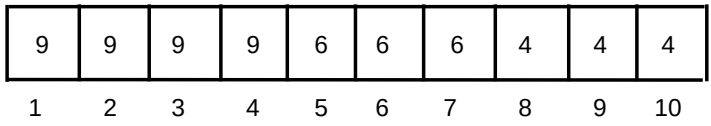

Figure 13: Example of array storing the number of jobs queued at LS level.

When there are more clusters with the same number of jobs with equal priority in the $P\text{-}th$ position, the one to dispatch a job is randomly selected.

3.2 Local-Scheduler

In this section we outline the algorithms we developed as Local-Scheduler: a Flexible Backfilling algorithm able to manage our specific constraints (Section 3.2.1), two different versions of a novel solution based on the Convergent Scheduling technique (Sections 3.2.2). In a first phase each algorithm is described, then we explain how they are integrated with our Meta-Scheduler to manage a set of distributed and heterogeneous resources organized as clusters located in a specific site.

The algorithms we propose to be used at LS level share a set of heuristics exploited to classify the submitted jobs. The heuristics are designed to manage specific problem constraints: job's QoS requirements, machine and licence usage. Each algorithm we developed exploits this set of heuristics, but heuristics are adapted to work according to specific algorithms needs, for instance: Flexible Backfilling does not exploit dynamic information about machines to classify jobs, instead, Convergent Scheduler uses some of these information in its job classification phase. Furthermore, the meaning of the heuristics results computed by the Flexible Backfilling is different with respect to that computed by the Convergent Scheduling technique.

The set of shared heuristics is composed by: *Anti-Aging*, *Deadline*, *Licences*, and *Wait Minimization*. Here, we describe the main structure of these heuristics, in the sections devoted to describe the proposed

solutions, we will specialize each of them in order to point out the main differences.

Anti-Aging. The goal of this heuristics is to avoid job starvation i.e. that a job could remains, for a long time, waiting to start or progress its execution. The heuristics computes a value Δ_{AA} as a function of the "age" that a job has reached in the system. Δ_{AA} is function of the difference between the $wall\_clock$ time and the time at which the job i is submitted to the LS. It is computed as:

$$\Delta_{AA} = (wall\_clock - Submission_i) \cdot age\_factor \;\; \forall i \in \textit{LS's queue}$$

where age_factor is the heuristics weight, $wall\_clock$ is the time at which the heuristics is computed, and $Submission_i$ is the time at which i is submitted to the scheduler. Anti-Aging is exploited in the same way, with the same meaning, in both Flexible Backfilling and Convergent Scheduler proposed solutions.

Deadline. The main goal of the Deadline heuristics is to maximize the number of jobs, which terminate their execution within their deadline. For each job i it requires an estimation of its execution time, $Estimated_i$, in order to evaluate its completion time with respect to the current $wall\_clock$. When the distance between the completion time and the deadline is smaller than a threshold value, the score Δ_D assigned to the job is increased in inverse proportion with respect to such distance. We define:

$$T_i \quad = \quad Deadline_i - k \cdot Estimated_i \tag{3.5}$$

where k is a constant value fixed by the system administrator. It permits us to overestimate $Estimated_i$. With $k = 1$, i.e. without overestimation, any event able to delay the execution of a job would lead to violate its deadline.

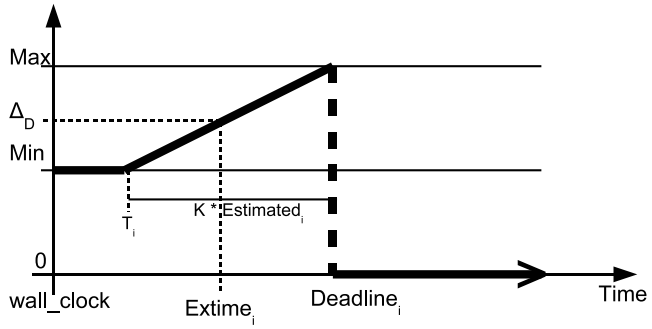

Figure 14: Graphical representation of the Deadline heuristics.

We define $Extime_i$ as the time at which LS estimates the termination of the job i. This parameter is computed as function of $Estimated_i$ in different ways according to the LS algorithm adopted, and it will be describe specifically in the next sections.

The heuristics aims to assign a minimum value Min to any job whose deadline is far from its estimated termination time. When the distance between the completion time and the deadline is smaller than a threshold value, T_i, the score assigned to the job is increased in inverse proportion with respect to such distance. The contribution of Deadline increases until $Extime_i$ reaches the job deadline, i.e. Δ_D is increased at least of Max score. When $Extime_i$ is greater than the job deadline, the contribution of the heuristics is null, i.e. Δ_D is set to 0 (Figure 14).

The Min and Max values are defined by system administrator. They are the weights of the heuristics and determine the angular coefficient a of the straight line passing through the points (T_i, Min), and $(Deadline_i, Max)$. Δ_D is increased according to the function described by the bold line in figure 14.

Licence. This heuristics favors the execution of jobs that increase the critical degree of licences. A licence becomes critical when there is a number of requests greater than the available number of its copies that can be

simultaneously activated. The Δ_L values carried out by Licences are computed as a function of the licence, critical and not critical, requested by a job. The greater the number of critical licences is, the greater Δ_L is. This pushes jobs using many licences to be scheduled first, this way releasing a number of licences, which can be assigned to other jobs waiting for execution.

To determine the critical licences, we introduce the ρ_l parameter defined as:

$$\rho_l = \frac{\sum_{i=1}^{|N|} s_{i,l}}{a_l} \;\; with\; s_{i,l} \in \{0,1\} \;\wedge\; s_{i,l} = 1 \text{ iff } i \in N \;\wedge\; i \text{ asks for } l \in L \quad (3.6)$$

where L is the set of all the available licences, the numerator specifies the "request", i.e. how many jobs are requesting a licence, and the denominator specifies the "offer", i.e. how many copies of a licence can be simultaneously activated on the system machines. A licence l is considered critical if $\rho_l > 1$. We divide licences according to their ρ value in two subsets:

$$L_c = \{l \in L \mid \rho_l > 1\} \quad (3.7)$$

$$L_{\bar{c}} = L \setminus L_c \quad (3.8)$$

L_c is the subset including critical licences, and $L_{\bar{c}}$ is the subset including not critical licences. Each algorithm adopted as LS exploits these subsets to compute Δ_L.

Wait Minimization. This heuristics favors jobs with the shortest estimated execution time. The rationale is that shorter jobs are executed as soon as possible in order to release the resources they need, and to improve the average waiting time of the jobs in the scheduling queue. Let $priority\_boost\_value$ be the heuristics weight defined by the system administrator according to system management policies, and $min\_ex\_t = min\{Estimated_i : i \in queue\}$, the value Δ_{WM} is computed by the heuristics as follows:

$$\Delta_{WM} += priority\_boost\_value \cdot \frac{min\_ex\_t}{Estimated_i}$$

3.2.1 Flexible Backfilling Algorithm

In this section we present the Flexible Backfilling algorithm (FB) we employed as Local-Scheduler (9). Our FB is a variant of the EASY Backfilling algorithm (35; 86). In general, we can recognize the structure of FB as the same as of our Meta-Scheduler (Section 3.1): it is characterized by a classification phase, in which jobs are labeled with a priority value, and by a scheduling phase, in which jobs are assigned to the available machines.

We developed FB in such a way it is able to classify incoming jobs in order to handle the jobs QoS requirements. Unlike MS, it uses dynamic information about both licences and jobs to carry out the job classification. Furthermore, it exploits dynamic information about jobs, licences, and machines to compute the scheduling plans. For instance, FB increases the job priorities according to the time they spent waiting for execution (i.e. according to the heuristics introduced previously).

The FB's classification phase is applied to each queued job at each scheduling event, which is job submission or job ending. In this way, the classification better represents the job QoS requirements with respect to the resource status, which could change over the time: job could be close to its deadline (or it could be not executed whiting its deadline), or a requested licence can become critical (or not critical). The main goal of the classification phase is to fulfill a set of users and system administrator QoS requirements.

FB exploits the set of previous described heuristics to assign a priority $P(i)$ to each job i: Anti-Aging, Deadline, Licence, and Wait Minimization. Anti-Aging and Wait Minimization are the general heuristics already presented, Deadline and Licence have been modified to be adapted to the specific scenario. The values computed by each heuristics are composed as a weighted sum, in which weights are set by system administrator for each heuristics:

$$P(i) = \Delta_{AA} + \Delta_D + \Delta_L + \Delta_{WM}$$

Deadline. This heuristics comes from that previously described. In particular, Deadline computes the $Extime_i$ value, i.e. the time at which FB estimates the termination of the job i, as follows:

$$Extime_i \quad = \quad wall\_clock + Estimated_i$$

where $Estimated_i$ is the estimated job execution time. Δ_D is computed by Deadline according to the following formula:

$$\Delta_D = \begin{cases} Min & \text{if } Extime_i \leq T_i \\ a(Extime_i - T_i) + Min & \text{if } T_i < Extime_i \leq Deadline_i \\ 0 & \text{if } Extime_i > Deadline_i \end{cases}$$

The Δ_D is set to a minimum value (Min) until the $Extime_i$ is smaller than T_i, it increases closing the job deadline, and it is set to 0 if the job termination is estimated after its deadline (Figure 14).

Licence. This heuristics assigns a higher score to jobs requiring a larger amount of critical licences. Δ_L is increased according to this formula:

$$\Delta_L = W \cdot (\sum_{l \in l_{\bar{c}}} \rho(l) + d \cdot \sum_{l \in l_c} \rho(l))$$

where W is the heuristics weight, and $d = max\{| \cup_{\forall i} l_{\bar{c}}(i)|, 1\}$ is the number of critical licences the job requires. This way we give more importance to critical licences computing the Δ_L value.

The scheduling phase is carry out applying the Backfilling strategy, which makes a resource reservation for the analyzed job if it cannot be scheduled at once, to a queue in which jobs are ordered according their

priority, i.e. the highest priority job is stored at the head of the queue. FB carries out a new scheduling plan at both job is submission and job ending time.

FB orders the machines it manages according to their computational power in order to exploit the most powerful first. In this way, even if Backfilling algorithms do not look for the best matching among jobs and machines (i.e. they assign a selected job to the first available machine able to run it), this strategy allows our FB to improve the response time of the jobs assigned to it. We also developed a procedure, called *Minimal Requirements*, that is able to selects a set of machines that has the computational requirements suitable to perform a job. In our study, we considered only the following requirements: number of processors and software licences activable on a machine. We apply the Backfilling strategy to this subset of the cluster's machines.

According to the problem description we gave in Section 1.2, we suppose that a set of software licences may be activated on the cluster's machines managed by FB. Each licence can be activated on a subset of the available machines, for instance, because the licence requires a specific operating system or a specific CPU. Moreover, the number of software licences of a specific type is generally smaller than the number of machines on which they can be activated. These are *floating* licences, because they can be activated dynamically on the proper machines according to the job requests. On the other hand, we define *non-floating* licences, which are permanently bounded on a specific machine, and that can be considered like any other attributes characterizing a machine. Each job requires a set of software licences for its execution and may be executed only on the subset of machines where all the required licences can be activated.

We designed two different variants of Flexible Backfilling: BF_UNMOD and BF_MOD. The first one keeps the reservation for the first queued job and it is preserved until the job is scheduled. The second one recomputes resource reservation when a new job reaches the first position in the waiting queue (i.e. it has the highest priority):

- BF_UNMOD implements a Flexible Backfilling strategy applied to a queue in which jobs are ordered according to their priorities. The

first queued job receives a resource reservation if it cannot be scheduled at once. It preserves the reservation (e.g. the highest priority) until its execution starts, while the others queued jobs are reordered according their priorities at each scheduling event. Like Easy Backfilling, BF_UNMOD adopts an "aggressive" strategy by enabling reservations for the first queued job only.

- BF_MOD differs from BF_UNMOD because it removes the reservation for the first queued job if a job with a higher priority is submitted. In this case, this new job receives the resource reservation if it cannot be scheduled at once. When a job i reaches the first position within the queue, it is allowed to reserve the resources it needs. Further, jobs are ordered according to their priorities and they can be used for backfilling. At the next scheduling event, the reservation made by i is preserved if and only if i still has the highest priority. Otherwise, the job with the highest priority is allowed to reserve resources. Suppose, for instance, that a job with a forthcoming deadline is submitted. BF_MOD schedules this job as soon as possible by canceling the resource reservation made for the first job in the queue at the next scheduling event. On the other hand, the prediction of the starting execution time of a job is difficult, and the risk is the job starvation. A simple way to reduce this phenomenon is to increase the weight computed by the Anti-Aging heuristics in the classification phase.

3.2.2 Convergent Scheduling Technique

The proposed Convergent Scheduling (CS) permits us to carry out a job-scheduling plan on the basis of the current status of the system (i.e. resource availability, executing jobs), and information related to jobs waiting for execution. In order to make decisions, the scheduler assigns priorities to all the jobs in the cluster (i.e. queued and running), and jobs priorities are computed at each scheduling event. The job priority measures the degree of preference of a job for each cluster's machine, i.e. how each machine suits well for the job execution. The scheduler aims to schedule

a subset of queued jobs that maximize the degree of preference for the available resources.

In order to exploit the CS technique, we define a matrix $P^{|N|\times|M|}$, called *Job-Machine* matrix, where N is the set of jobs (it changes dynamically) and M is the set of the cluster's machines. The matrix entries store a priority value specifying the degree of preference of a job for each cluster's machines. Our scheduling framework is structured according to three consecutive phases: Heuristics, Clustering and Pre-Matching, and Matching.

The Heuristics phase is computed to assign a priority to each job in the system. The Job-Machine matrix constitutes the common interface to each heuristics we defined: Minimal requirements, Deadline, Licences, Input, Overhead minimization, Wait minimization and Anti-Aging. Each heuristics changes Job-Machine entries to increase/decrease the degree of matching between a job and a machine. The priority values are computed at each scheduling event. Moreover, the priority value stored in each matrix's entry is the weighted sum of the values computed by each heuristics. Heuristics can be run several times and in any order, moreover, each heuristics manages a specific problem constraint and the heuristics set is easy to extend.

The Clustering & Pre-Matching phase takes as input the Job-Machine matrix and it aims at removing conflicts on the licence usage: the number of activated copies of each licence must not be greater than the number of its available copies. The first step of this phase is to cluster jobs according to their requests for licences. Then, the Multidimensional $0-1$ Knapsack Problem algorithm (MKP) (71) is applied to find the subset of jobs that can be simultaneously executed without violating the constraints on the licence usage. Jobs discarded by this phase are then reconsidered to build the new scheduling plan, at the next scheduling event.

Matching elaborates the matrix resulting from the previous phase to carry out the job-machine associations, i.e. the new scheduling plan. The aim of the Matching phase is to compute the job-machine associations to which correspond a larger preference degree, according to the problem constraints.

Heuristics Collection

Seven heuristics have been designed to build our Convergent Scheduling framework: Anti-Aging is the general heuristics presented at the beginning of section 3.2, Deadline, Licences and Wait Minimization comes from the ones described previously, and Minimal requirements, Overhead minimization and Input have been designed specifically for the Convergent Scheduler. The order of magnitude of the heuristics complexity is equal to $(|N| \cdot |M|)$.

Minimal requirements. This heuristics fixes the associations job-machines. It selects the subset of machines M_{Aff_i} that has the computational requirements suitable to perform a job i. In our tests, we considered only two requirements: number of processors and floating licences (i.e. number and type).

Deadline. The aim of the Deadline heuristics is to compute the job-machine update value Δ_D in order to execute jobs respecting their deadline. This heuristics differs from that employed in our Flexible Backfilling because of the job deadline is evaluated with respect to all the machines in M_{Aff_i}. This means that the priority of a job-machine association (i.e. an entry in the Job-Machine matrix) is improved if that machine is able to perform the job respecting its deadline. For each job $i \in N$ the Δ_D values are computed with respect to the machines $m \in M_{Aff_i}$. We define:

$$f_1 = \begin{cases} Min & \text{if } Extime_{i,m} \leq T_{i,m} \\ a(Extime_{i,m} - T_{i,m}) + Min & \text{if } T_{i,m} < Extime_{i,m} \leq deadline_i \\ Min & \text{if } Extime_{i,m} > deadline_i \end{cases}$$

Min and Max, in Figure 15, have the same meaning of the general Deadline heuristics. $T_{i,m}$ is the time from which the job must be evaluated to meet its deadline if it is performed by the machine m (i.e. T_i in the general expression (3.5)). The $Nxtime_{i,m}$ is used to compute $T_{i,m}$ and $Extime_{i,m}$ as follows:

$$
\begin{aligned}
T_{i,m} &= Deadline_i - k \cdot Nxtime_{i,m} \\
Extime_{i,m} &= wall\_clock + Nxtime_{i,m} \\
Nxtime_{i,m} &= (estimated_i \times (1 - progress_i)) \cdot \frac{BM_{\bar{m}}}{BM_m} \qquad (3.9)
\end{aligned}
$$

$Nxtime_{i,m}$ estimates the execution time of the job i with respect to the estimated execution time specified by i ($estimated_i$), the machine on which i is performed, and the percentage of i already executed ($progress_i$). In the expression (3.9), BM_m is the power of the machine m where i is executing, and $BM_{\bar{m}}$ is the power of the machine m utilized to estimate the execution time of i. This time can be evaluated statistically by analyzing historical data or by benchmarking.

The values Δ_D to update the Job-Machines matrix entries are computed according to the following expressions:

$$
\begin{aligned}
\Delta_D &= (\sum_{k=1}^{|M_{Aff_i}|} f_1(Extime_{i,k})) \cdot \frac{\bar{f}_1(Extime_{i,m})}{d_i} \qquad (3.10) \\
\bar{f}_1(Extime_{i,m}) &= Max - f_1(Extime_{i,m}) \\
d_i &= \sum_{m=1}^{|M_{Aff_i}|} \bar{f}_1(Extime_{i,m})
\end{aligned}
$$

The first term of (3.10) establishes the job "urgency" to be executed respecting its deadline. A higher value of $f_1(Extime_{i,m})$ for a job means that it is a candidate (i.e $Extime_{i,m}$ is very close to $Deadline_i$) to complete its execution missing its deadline when executed on m. High values of the first term mean that the related job could be in "late" on all or almost all the available machines. Therefore, the priority of such job has to be increased more than those of jobs that have obtained lower values. The second term of (3.10) distributes on the available machines the values computed by the first term, proportionally to machine power. For each job $\bar{f}_1(Extime_{i,m})$ establishes a ordering of the machines w.r.t. their power, on which it can be performed. The higher the $\bar{f}_1(Extime_{i,m})$

value associated to a machine, the higher the probability that such machine will perform the associated job respecting its deadline (Figure 15).

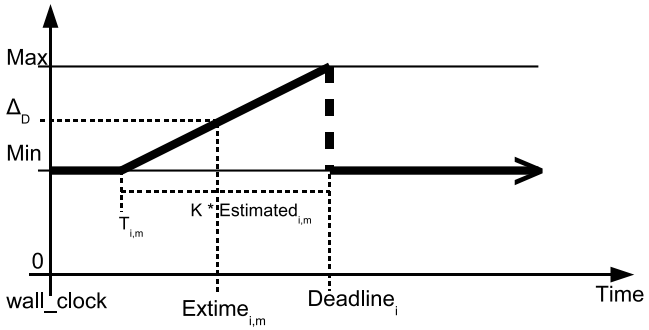

Figure 15: Graphical representation of the CS Deadline heuristics.

Licences. This heuristics favors the execution of jobs that increase the critical degree of licences. The Δ_L values to update the Job-Machine matrix entries are computed as a function of the licences, critical and not critical, requested by a job. The greater the number of critical licences is, the greater Δ_L is. Referring to the licences subsets defined by expressions (3.7) and (3.8), we define $w(i)$ for the job i as follow:

$$w(i) = \sum_{l=1}^{|L_c|} s_{i,l} \cdot \rho_l \cdot |L| + \sum_{l=1}^{|L_{\bar{c}}|} s_{i,l} \cdot \rho_l$$

where $|L|$ is used as a multiplicative constant to give more weight to critical licences, ρ_l and $s_{i,l}$ is computed according to the expression (3.6).

We order jobs according to $w(i)$. Let $a, b \in N$ two jobs, and

$$
\begin{aligned}
w_1(i) &= \sum_{l=1}^{|L_c|} s_{i,l} \cdot \rho_l \\
w_2(i) &= \sum_{l=1}^{|l_{\bar{c}}|} s_{i,l} \cdot \rho_l \\
a < b \quad \text{iff} \quad & \begin{cases} w_1(a) < w_1(b) \ \text{ with } \ w_1(a) \neq w_1(b) \\ w_2(a) < w_2(b) \quad otherwise \end{cases}
\end{aligned}
$$

Δ_L is computed as:

$$\Delta_L = LI \cdot \left(\frac{w(i)}{max(w(i))} \right) \ \ \forall i \in N$$

where LI is a value used to fix the weight of the heuristics.

Input. The goal of this heuristics is to update the matrix entries according to the cost due to the transfer of the job input data on the machines candidate to run it. Such data can be the original job input data, also distributed across more machines, or partial results carried out by the job itself before an interruption. Since we cannot do any assumption about how jobs use the input data, a reasonable criteria is to try to execute a job on the machine to which correspond to the minimum data transfer time. The input data transfer time for a job i is computed as:

$$Transf_i(m) = \sum_{\forall m \in M_{Aff_i} \wedge \bar{m} \neq m} \frac{input_{i,\bar{m}}}{b_{\bar{m},m}} \ \text{ with } i \in N$$

where m is the machine candidate to run i, $\bar{m}$ is the machine on which the input is stored, $input_{i,\bar{m}}$ is the data size, and b is the bandwidth of the link connecting $\bar{m}$ and m.

The updated value is computed as an inverse proportion of the data transfer time:

$$\Delta_I = IN \cdot \left(1 - \frac{Transf_i(m)}{\sum_{\forall m \in M_{Aff_i} \wedge \bar{m} \neq m} Transf_i(m)} \right)$$

where IN is heuristics weight fixed by the installation.

Wait Minimization. The aim of this heuristics is to minimize the average time that jobs spend waiting to complete their execution. It tries to minimize this time by advancing the jobs that have a shorter estimated time to end their execution. Δ_{WM} is computed as a function of the time remaining to end the execution of a job i on each machine $m \in M_{Aff_i}$. The smaller this time is, the greater the value to update the Job-Machine matrix is, and it is computed as:

$$\Delta_{WM} = W \cdot \left(1 - \frac{Nxtime_{i,m}}{sup_{Nxtime}}\right)$$

where max is the heuristics weight fixed by the installation, $Nxtime_{i,m}$ is computed according to the expression 3.9, and sup_{Nxtime} is set equal to the maximum value of $Nxtime_{i,m}$ for each $m \in M_{Aff_i}$.

Overhead Minimization. The heuristic aims to contain the overhead due to the interruption of a job execution on a machine and its resuming on another one, or on the same one at a different time. This heuristic checks the current job-machine matching and tries to minimize the job migrations by preserving the scheduling made in the previous step.

Clustering and pre-matching

This phase is executed after the Job-Machine matrix has been updated by all the heuristics. All jobs requesting critical sw licences (i.e. $\rho_l > 1$) are clustered by putting in the same cluster the jobs asking for the same licence/s, and with each job belonging to only one cluster.

A new event (job submission or ending) can change the licence usage. When a licence becomes critical, two or more clusters could be merged. When a licence becomes not critical, a cluster could be partitioned into two or more new clusters. It is implemented by modeling the requests of sw licences as a hypergraph where the nodes represent the licences and the arcs the jobs.

The MKP (Multidimensional 0-1 Knapsack Problem) optimization method (71) is applied to each cluster to find the subset of jobs that could be simultaneously executed, without violating the constraint on the licences usage. The remaining subset will be discarded, i.e. their entries are deleted from the job-machine matrix. The resulting Job-Machine matrix will be passed to the Matching phase to carry out the job-machine associations, i.e. the new job scheduling plan. In our case, the knapsack's capacities are the number of licences of each specific type simultaneously usable, and the objects are the jobs that require such licences.

Defining the costs c_i and the weights a_i of a job i as follows, the solution proposed by Eilon (**?**) can be exploited to formulate the MKP as a KP.

$$\bar{c}_i = \frac{\sum_{m=1}^{M} p_{i,m}}{|\{p_{i,m} \in P : p_{i,m} > 0\}|}, \quad \bar{a}_i = \sum_{l=1}^{|L_c|} \frac{a_{i,l}}{b_l}$$

where L_c is the set of critical licences, b_l is the availability of the l-th critical licence, and $a_{i,l}$ is equal to 1 iff the job i uses the licence l.

To solve the resulting KP we choice a greedy algorithm with polynomial complexity, which adopts the decreasing unitary costs method (3.11) as the ordering heuristic to select the objects.

$$\frac{\bar{c}_1}{\bar{a}_1} \geq \frac{\bar{c}_2}{\bar{a}_2} \geq \ ..... \ \geq \frac{\bar{c}_n}{\bar{a}_n} \tag{3.11}$$

According to the decreasing unitary costs ordering, jobs can be discarded. By discarding a job some licences could become not critical. To avoid the exclusion of jobs that can be executed by exploiting these licences, the job weights are recomputed after each job discarding. The greedy algorithm was validated by comparing its results with those obtained by a branch&bound algorithm.

Matching

The aim of this phase is to carry out the best job-machine associations, i.e. the new job scheduling plan. Starting from the Job-Machine matrix

P, resulting from the previous phases, it looks for a matching $MTC \subset N \times M$ which corresponds to the largest preference degree, according to the constraint that each machine must perform one job at time, and that a job can be scheduled only on one machine. The best MTC is the one that maximizes both the cardinality of the job-machine associations, i.e. $|MTC|$, and the sum of the selected associations, i.e. $\sum p_{i,m} \in MTC$. The maximum system usage is obtained when $|M| = |MTC|$. Formally, we need to find a MTC that maximize:

$$\max \sum_{0 \leq i < |N|} \sum_{0 \leq m < |M|} p_{i,m} \cdot x_{i,m}$$

where $p_{i,m} \in P$, and with the following constraints:

$$\begin{cases} \forall m & \sum_{i=1}^{N} x_{i,m} \leq 1 \textit{ maximum a job for machine} \\ \forall i & \sum_{m=1}^{M} x_{i,m} \leq 1 \textit{ maximum a machine for a job} \\ \forall i, \forall m & x_{i,m} \in \{0, 1\} \end{cases}$$

where $x_{i,m} = 1$ if m is chosen to run i, otherwise $x_{i,m} = 0$.

Different methods could be used to solve this problem according to a trade-off between accuracy and running time. In this work we investigated three different methods: Maximum Iterative Search (MIS), Maximum Flow Search (MFS), and Incremental Maximum Flow Search (IMFS). The complexity of the corresponding algorithms was computed fixing the size of P equal to $|N| \times |N|$ (i.e. $|M| = |N|$).

MIS was implemented by adopting a greedy algorithm with complexity $O(|N|^2 \, log \, |N|^2)$, which carries out the job-machine associations by selecting at each iteration the one with maximum cost. MIS does not ensure to find a maximum matching cardinality, i.e. it does not assure the maximum system usage.

MFS represents the matrix P as a bipartite graph $G = (O \cup D, A)$ where $O = \{1, \ldots |N|\}$ is the set of nodes representing jobs, $D = \{N + 1, \ldots N + |M|\}$ is the set of nodes representing the machines, and $A \subseteq O \times D$, with $|A| = |M|$, is the set of arcs. The arcs represent the job-machine associations, and have an associated cost $c_{i,m}$. For each element $p_{i,m}$ of

the Job-Machine matrix, an arc is created between the node i representing a job and a node j representing a machine, according to the following expression:

$$\forall i, \forall m \quad \left(p_{i,m} > 0 \Rightarrow \exists (i,m) \in A \wedge c_{i,m} = p_{i,m}\right)$$

When an association job-machine is not eligible, i.e. at least a constraint is violated, the related matrix entry is set equal to zero. In other words, we look at the Job-Machine matrix as a bipartite graph, with arcs weighted proportionally to the matrix entries, and we try to find the association set that maximizes the cost (sum of the arc weights). This way we match jobs and machines with the highest preference.

MFS incrementally builds the final matching passing through partial matching built exploiting the Longest Path Tree (LPT), which is computed starting from all the nodes that are not associated (i.e. nodes do not present in the current partial matching). As a result we obtain a tree where nodes represent a job or a machine, and where each leaf has a value representing the cost (i.e. the sum of the weights associated to the arcs pass through) to reach it. On this tree the largest cost path is selected. The new matching is found by visiting bottom up the selected path and by reversing the direction of the crossed arcs. We have two kinds of associations. Associations corresponding to reverse arcs with negative cost and associations corresponding to forward arcs, with positive cost. The first ones are deleted from the previous matching, while the second ones are inserted into the new matching plan.

The MFS solution we propose permits us to carry out a job-machine association set with max cardinality and max cost, and with a complexity equal to $O(N^4)$. This is feasible if the algorithm is fast enough to provide fresh data for the scheduling. In our case, the set of jobs is dynamic, so the MFS costly algorithm was developed into IMSF in order to reduce the complexity of this phase.

IMSF is able to carry out a new scheduling plan exploiting partial results coming from the previous one. Instead of rebuilding the bipartite graph from scratch, it starts from a partial bipartite graph built exploiting the job-machine associations inherited by the previous matching phase.

IMFS builds a new bipartite graph changing the way and the sign of the weight, of the arcs, related to the inherited job-machine associations. Then, on the graph, the LPT is computed. On this tree we can distinguish two kinds of paths: *even* and *odd*, w.r.t. the number of arcs passed through. The first ones must end in a node representing a job, while the odd ones must end with a $machine$ not yet associated to a job. When possible, odd paths are preferred to even paths because they lead to an improvement of the matching cardinality. Then, the path with the largest cost is selected, i.e. the one that leads to the largest increase of the matching size. The IMSF complexity is equal to $O(N^3)$, it is an approximation of MFS.

Chapter 4

Experiments

In this Chapter we present the experiments we performed to evaluate our solutions. The Multi-Level scheduling framework, i.e. Meta-Scheduler (MS) and Local-Scheduler (LS), the Flexible Backfilling, and the Convergent Scheduler were evaluated by simulation to verify their feasibility.

In our study we consider a continuous stream of independent batch jobs, which arrive to the system and are stored into a single job queue. We suppose that a job is sequential or multi-threaded and that a job is executed only on a machine. For each experimental phase we point out if the space sharing allocating policy is adopted, and if job preemption is allowed. We also assume that mechanisms to notify configuration changes, such as job submission/ending, are available in the computing platform.

To evaluate the Multi-Level Scheduler, we simulate a dedicated computing grid, composed of independent clusters of heterogeneous, single-processor or SMP machines, linked by a low-latency and high-bandwidth network. Each cluster includes machines located in a specific site. To evaluate the Flexible Backfilling and the Convergent Scheduler we simulate only a cluster of heterogeneous machines.

The schedulers aim to schedule arriving jobs respecting their QoS requirements (i.e. job deadlines, licence requirements, and user peculiarities), and optimizing the software licence and machine usage.

Submitted jobs and machines are annotated with information de-

scribing computational requirements and hardware/software features, respectively. Each job is described by an identifier, its deadline ($Deadline$), an estimation of its duration ($Estimated$), a benchmark score ($Benchmark$), which represents the architecture used to estimate its execution time, and the number of processors (CPU) and software licences requested ($Licence\ needs$) to run it. Machines are described by a benchmark score, which specifies their computational power, and the number of available CPUs. For each test we specified how these parameters are generated.

For our evaluation, we developed an event-based simulator, where events are job arrival and termination. We simulated a computing farm, with varying number of jobs, machines and licences availability. For each simulation, we randomly generated a list of jobs and machines. We exploited streams of jobs synthetically generated because real workload traces with deadlines and software requirements are not easy to find. We studied the real workload traces proposed by Feitelson in (36), and the data provided by the Italian Interuniversity Consortium (25). In both cases any requirement about deadlines and licences are provided for jobs.

A simulation step includes: (1) selection of new jobs, (2) update of the status (e.g. the job execution progress) of the executing jobs, (3) check for job terminations. The time of job submission is driven by the $wall\_clock$. When the $wall\_clock$ reaches the job submission time, the job enters in the simulation.

4.1 Flexible Backfilling

In this section we present the evaluation conducted to investigate the effectiveness of the scheduling solutions carried out by our Flexible Backfilling scheduler (Section 3.2.1). The evaluation was conducted by simulations using different streams of jobs which inter-arrival times are generated according to a negative exponential distribution with a different parameter. For each simulation, we randomly generated a list of jobs and machines whose parameters are generated according to a uniform distribution in the ranges described in Table 1, where $Licence\ needs$ specifies

Table 1: Flexible Backfilling: range of values used to generate streams of jobs, machines and sw licences.

Parameters	Jobs	Machines	Sw Licences
Estimated	[500 - 3000]		
Benchmark	[100 - 500]	[100 - 500]	
Margin	[30 - 250]		
CPU	[1 - 8]	[1 - 8]	
Licence needs	[30%]		
Licence ratio			[50% - 70%]
Licence Suitability			[90%]

the probability that a job needs a licence, the $Licence\ suitability$ specifies the probability that a licence can be used on a machine, and $Licence\ ratio$ specifies the maximum number of copies of a licence concurrently usable with respect to the $Licence\ suitability$.

Tests were conducted by simulating a cluster of 100 machines, 20 different types of software licences, 1000 not preemptable, independent jobs, and using 5 job streams generated with average jobs inter-arrival time fixed equal to 4, 6, 12, 24 and 48 simulator time units. Each stream leads to a different system workload (computed as the sum of the number of jobs ready to be executed, and the number of the jobs in execution) through a simulation run. The closer job inter-arrival time is, the higher the contention in the system is. The space sharing job allocating policy is not used in these experiments. To obtain stable values each simulation was repeated 20 times with different job attributes values. The classification heuristics are computed at each scheduling event[1].

To evaluate the schedules carried out by the two different variants of Flexible Backfilling we developed (Section 3.2.1), we have considered the following metrics:

- System Usage. This measures the efficiency of the system, and it is

[1]Heuristics weights: Deadline ($MAX = 20.0, MIN = 0.1, k = 1.4$), Anti-aging ($age\_factor = 0.01$), Licences ($W = 1$), Wait minimization ($priority\_boost\_value = 2.0$).

defined as follows:

$$System\_Usage = \frac{\sharp CPU\_in\_use}{min(\sharp total\_CPUs, \sharp jobs\_in\_system)}$$

where $\sharp CPU\_in\_use$ is the number of CPUs executing a job, $\sharp total\_CPUs$ is the available total number of CPUs, and $\sharp jobs\_in\_system$ sums the number of waiting jobs and those in execution.

- Out Deadline. This measures the number of jobs executed without respecting their deadline. This does not include jobs which must not be executed within a given deadline.

- Slowdown. This measures the ratio between the response time of a job, i.e. the time elapsed between its submission and its termination, and its execution time.

We have compared BF_MOD and BF_UNMOD with FCFS and with BF_FCFS, which is an implementation of EASY backfilling. The implementation of our extend versions of Backfilling differ with respect to the classical algorithm because of the target architecture which is an heterogeneous architecture rather than a homogeneous multiprocessor machine. As a consequence, jobs considered by our algorithms may require different software/hardware resources. The original Backfilling algorithms have been modified to consider all these resources when they carry out a scheduling plan.

In Figure 16 and Figure 17 the results obtained for the different strategies with respect to the metrics previously defined are compared. Figure 16 shows the percentage of the jobs executed missing their deadline. It can be seen that BF_MOD and BF_UNMOD obtain better results in each test. When the available computational power is able to maintain low the system contention (i.e. for 24, 48 average job inter-arrival times), the use of Backfilling technique leads to a higher system usage, which permits to improve the percentage of the jobs that are executed respecting their deadline. On the other hand, when the system contention is higher (i.e.

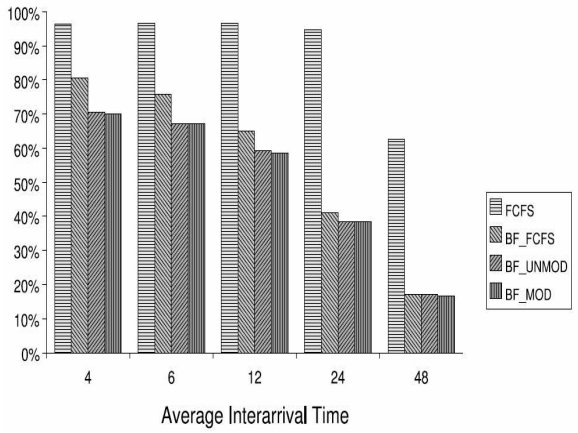

Figure 16: Percentage of the jobs executed that miss their deadline.

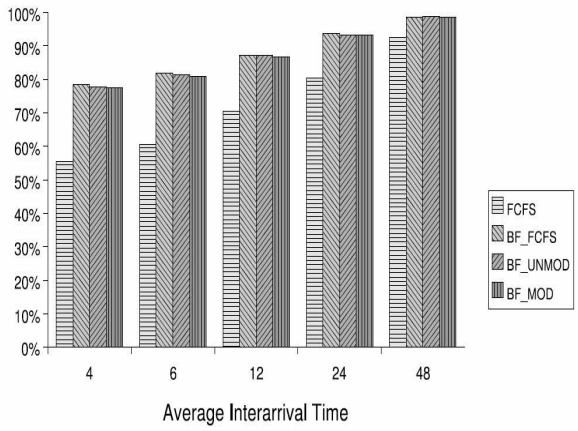

Figure 17: (Percentage of used system hardware resources.

for 4, 6, 12 average job inter-arrival times) the exploitation of the job priority leads to better results. Figure 17 shows the percentage of system usage. It can be seen that Backfilling technique leads to a better system usage, in particular when the system contention is higher.

Figure 18 shows the slowdown trend through simulation runs. It can

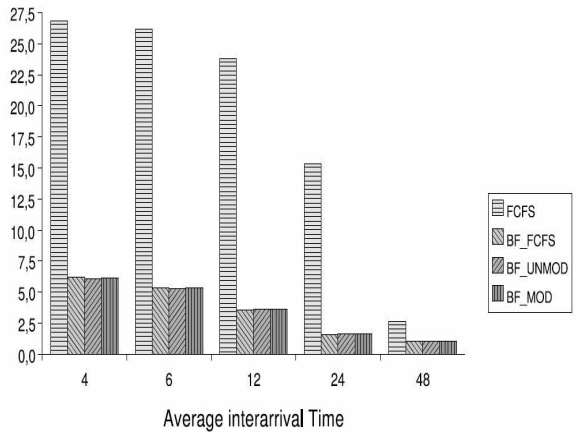

Figure 18: Slowdown trend.

Table 2: Percentage of used licences.

Licence Ratio	**% Licence Usage**			
	FCFS	BF_FCFS	BF_UNMOD	BF_MOD
30% - 50%	55,7	78,5	80,5	79,7
40% - 60%	52,1	76,1	75,5	76,3
50% - 70%	49,5	74,8	72,2	72,4

be seen that the Backfilling technique is able to drastically reduce the average job waiting time.

Table 2 and Table 3 show the percentage of both the software licence usage and the number of jobs executed out of their deadline by changing the $Licence\_Ratio$ parameter.

Table 2 shows that to assign higher priorities to jobs requiring a higher number of critical software licences leads to an improvement only when the software licences contention is high. When the software licences contention decreases the proposed schedulers lead to worse results. This because jobs requiring a fewer number of critical licences, but with a closer deadline, receives higher priorities delaying the execution of jobs requiring a higher number of software licences but with a far deadline.

Table 3: Percentage of jobs executed that miss their deadline.

Licence Ratio	% of jobs out of deadline			
	FCFS	BF_FCFS	BF_UNMOD	BF_MOD
30% - 50%	95,6	78,9	74,5	73,5
40% - 60%	95,5	74,8	69,4	69,0
50% - 70%	95,5	74,6	66,2	66,2

Table 3 shows that, when the contention on software licence increases, the scheduler changing the reservation for the first queued job at each job scheduling event permits us to obtain a higher number of jobs executed respecting their deadlines.

The experimental results show the applicability of the proposed strategy. Both BF_UNMOD and BF_MOD outperforms FCFS and BF_FCFS in term of system usage and in the number of jobs that are scheduled respecting the proposed deadline. The differences are strong at any load level (i.e. job inter-arrival time). With higher inter-arrival times, i.e. when the scheduling is less critical, BF_FCFS shows a performance similar to our two strategies.

BF_UNMOD and BF_MOD do not greatly improve the slowdown over BF_FCFS, which already does a very good scheduling job with respect to standard FCFS. We were not able to measure any difference between BF_UNMOD and BF_MOD. This means that the simpler approach followed by BF_UNMOD is sufficient for the task at end.

4.2 Convergent Scheduler

In this section, we present the results of two different evaluation phases. The first one was conducted to evaluate the quality of the solutions carried out by the MKP algorithm. The second one has verified the feasibility of Convergent Scheduling (CS). We evaluated four different versions of CS, called CS_i_p, CS_i_np, CS_ni_p, CS_ni_np. CS_i_p uses the incremental approach and preemption for all jobs. CS_i_np uses the incremental approach, but not preemption. CS_ni_p uses the non incremental approach,

Table 4: Convergent Scheduler: range of values used to generate streams of jobs, machines and sw licences.

Parameters	Jobs	Machines	Sw Licences
Estimated	[500 - 3000]		
Benchmark	[200 - 600]	[200 - 600]	
Margin	[25 - 150]		
CPU	[1 - 8]	[1 - 8]	
Input	[100 - 800]		
Licence needs	[20%]		
Licence ratio			[55% - 65%]
Licence Suitability			[90%]

and preemption for all jobs. CS_ni_np uses the non incremental approach, and not preemption.

In this work, all CS versions have used the same set of heuristics, the parameters of which were hand-tuned to give more importance to the job deadline requirement and the licence usage constraint[2]. We simulate job that are non preemptable by increasing the priority of the jobs being executed of a value higher than that used to increase the priority of the other jobs. In such a way, previous scheduling choices are preserved, and a scheduling plan is changed when a job ends.

The evaluation was conducted by simulations using different streams of jobs generated according to a negative exponential distribution with different inter-arrival times between jobs.

For each simulation, we randomly generated a list of jobs and machines whose parameters are generated according to a uniform distribution in the ranges described in Table 4.

To each licence we associate the parameter $Licence\ ratio$ that specifies its maximum number of copies concurrently usable with respect to the $Licence\ suitability$, which specifies the probability that a licence is usable on a machine. $Licence\ needs$ specifies the probability that a job needs a

[2]Heuristics constant values used in our tests: Deadline ($MAX = 10.0, MIN = 0.1, k = 1.3$), Input ($IN = 3.0$), Overhead Minimization ($\Delta = 8.0$ for CS version using job preemption, and $\Delta = 800.0$ for CS version not using job preemption), Anti-aging ($age\_factor = 0.02$), Licences ($LI = 2.5$), Wait minimization ($W = 4.0$).

licence. In Table 4, $Input$ is the range identifying the size of input data requested by jobs.

For each simulation, 30% of jobs were generated without deadline. In order to simulate the job scheduling and execution, the simulator implements the heuristics seen before, and a new scheduling plan is computed at the termination or submission of a job. The simulation ends when all jobs are elaborated.

To evaluated the CS schedulers, tests were conducted by simulating a cluster of 150 machines, 20 different types of licence, and 1500 jobs. To evaluate the MKP algorithm, in order to save simulation time, 100 machines, 40 licences, and 1000 independent jobs were used.

In order to obtain stable values, each simulation was repeated 20 times with different job attributes values.

To evaluate the solutions carried out by the MKP algorithm we compared them with those computed by a branch& bound algorithm.

To evaluate the quality of schedules computed by CS we exploited different criteria: the percentage of jobs that miss the deadline constraint, the percentage of machine usage, the percentage of licence usage, and scheduling computational time. The evaluation was conducted by comparing our solution with EASY Backfilling (BF-easy), Flexible Backfilling algorithms, and Earliest Deadline First(EDF). The Flexible Backfilling algorithms we adopted are those described in the section 3.2.1, and evaluated in the previous section. The job priority values are computed at the computation of a new scheduling plan using the CS heuristics. Computing the priority value at each new scheduling event permits us to better meet the scheduler goals.

To implement EDF, jobs were ordered in decreasing order with respect to their deadline, and machines were ordered in decreasing order with respect to the value of the $Benchmark$ attribute. Each job was scheduled to the best available machine, respecting the constraints on both the number of CPUs and of licences. If an eligible machine is not found the job is queued and re-scheduled at the next step of the simulation process. CS exploits the job preemptive feature to free resources to execute first new submitted job with an earliest deadline.

Table 5: Results obtained by running the algorithms branch&bound and MKP on the same job streams instances.

Licence ratio	**branch&bound**			
	Time (ms)	Sml (%)	Avg (%)	Bst (%)
$[40-45]$	496.53	0.57	0.87	1.00
$[45-50]$	353.51	0.62	0.88	1.00
$[50-55]$	331.28	0.64	0.89	1.00
$[55-60]$	556.96	0.64	0.88	1.00
Licence ratio	**MKP**			
	Time (ms)	Sml (%)	Avg (%)	Bst (%)
$[40-45]$	11.23	0.54	0.82	1.00
$[45-50]$	10.40	0.59	0.84	1.00
$[50-55]$	10.42	0.62	0.87	1.00
$[55-60]$	11.90	0.63	0.87	1.00

To evaluate the quality of the solutions carried out by the MKP algorithm, tests were conducted generating instances of job streams varying the value of the $Licence\ ratio$ parameter in the ranges $[40\% - 45\%]$, $[45\% - 50\%]$, $[50\% - 55\%]$, $[55\% - 60\%]$, and fixing the job inter-arrival times at 10 simulator time unit. Table 5 shows the results obtained by running the branch&bound and MKP algorithms on the same job streams instances. For each instance and algorithm, the Table reports: the smallest (Sml), the best (Bst) and the average (Avg) percentage value of licence usage, and the average algorithm execution time (Time).

As expected, the smaller the contentions are (i.e. for higher $Licence\ ratio$ value), the greater the probability that MKP carries out solutions closer to the optimal one is. MKP is able to find solutions that are on average 3% distant from those found by the branch&bound algorithm, by saving 97% of CPU time.

To evaluate the schedules carried out by the CS schedulers, we generated seven streams of jobs with jobs inter-arrival times (Ta in Figure 19) fixed equal to 4, 6, 8, 10, 12, 16 and 20 simulator time unit ($4 \div 20$ in

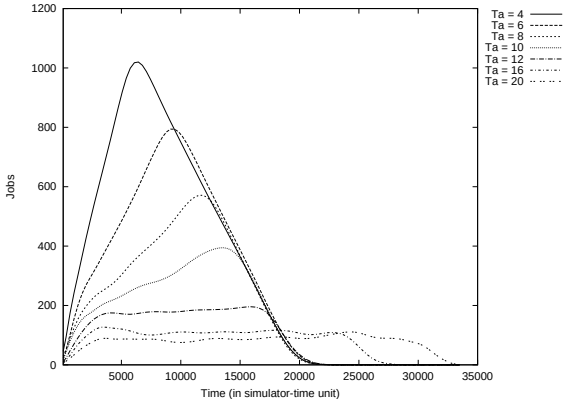

Figure 19: System workload through simulations.

Figures 20, 21, 22, 23). As shown in Figure 19, each stream leads to a different system workload through a simulation run. The system workload was estimated as the sum of the number of jobs ready to be executed plus the number of jobs in execution. The shorter the job inter-arrival time is, the higher the contention in the system is. As it can be seen, when the jobs inter-arrival time is equal or greater than 12 simulator-time units the system contention remains almost constant through the simulation: this is because the cluster computational power is enough to prevent the job queue from increasing.

We first measured the percentage of jobs executed that do not respect their deadline. Figure 20 shows this value for the seven job stream distributions. As expected, the smaller the job inter-arrival time is, the greater the job competition in the system is, and consequently the number of late jobs improves. Satisfactory results are obtained when the available cluster computational power is able to maintain almost constant the system contention. The three backfilling algorithms and EDF obtain worse results than those obtained by all the CS schedulers. Moreover, CS_ i_p and CS_ ni_p obtain equal or very close results demonstrating that the CS incremental version can be used without loss of results quality.

Figure 21 shows the slowdown parameter evaluated for jobs without

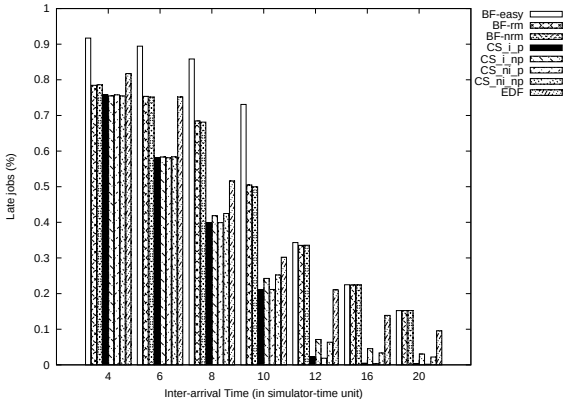

Figure 20: Percentage of jobs executed missing their deadline.

the deadline. This parameter gives us as the system load delays the execution of such jobs. As it can be seen, for job streams generated with a inter-arrival time equal or greater than 10 simulator time unit, jobs scheduled by using a CS scheduler obtained a Slowdown equal or very close to 1.

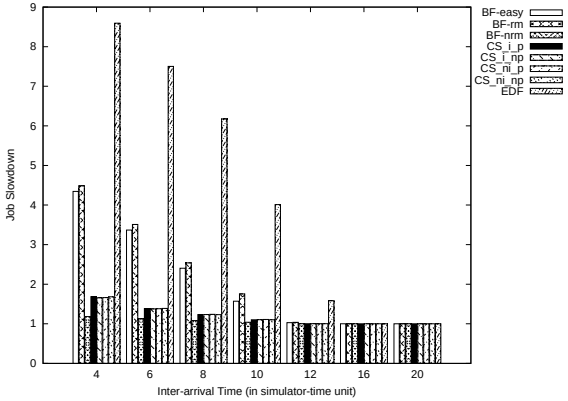

Figure 21: Slowdown of jobs submitted without deadline.

In Figure 22, the percentage of machine usage is shown. Since the

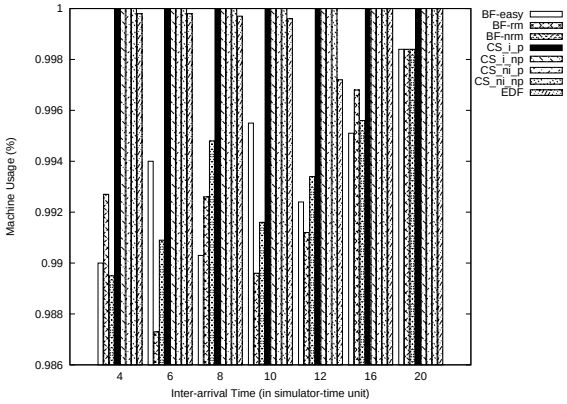

Figure 22: Percentage of machine usage.

considered algorithms do not support the processor space sharing allocation policy, this measure permits us to roughly figure out the system utilization. It is obtained averaging out the value:

$$\frac{\#\ of\ active\ machines}{min(\#\ of\ available\ machines, \#\ of\ jobs\ in\ the\ system)}$$

computed at each instant of a simulation. A machine can remain idle when it does not support all the hardware/software requirements required by a queued job. We can see the CS algorithms schedule jobs in a way that permits to maintain all the system machines always busy.

In Figure 23, we show the average scheduling times spent by the schedulers for conducting the tests on the simulated computational environment. As expected, the CS versions which support the preemption feature require more execution time. CS_ i_p and CS_ i_np obtain a lower execution time than CS_ ni_p and CS_ ni_np, respectively.

In Figure 24 we show the scalability for the CS and EDF algorithms. The CS scheduler scalability was evaluated for the versions using the incremental approach (CS-incr) and non using the incremental approach (CS-noincr) both using the job preemption feature. It was evaluated measuring the time needed to carry out a new scheduling plan increasing the

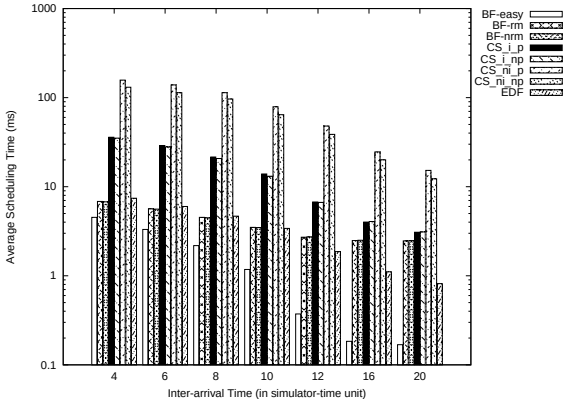

Figure 23: Average scheduling times.

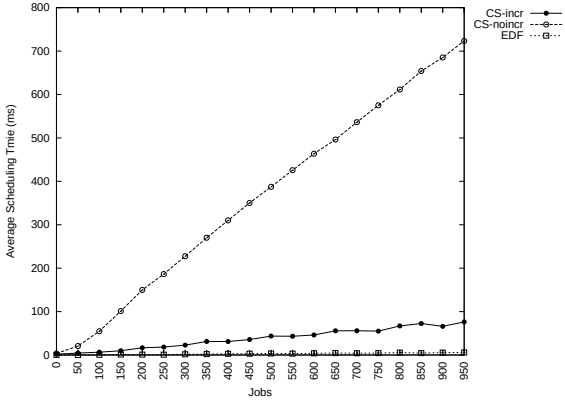

Figure 24: Algorithm scalability.

number of jobs. It can be seen that the CS-incr versions obtained very good scalability results.

4.3 Multi-Level Framework

In this section we present the evaluation of the scheduling solution carried out by the proposed two-level scheduler. To evaluate the proposed solutions we exploited four metrics:

- Percentage of workload elaborated by each cluster. It shows in which way the MS scheduling policies work.
- Percentage of jobs that miss their deadline. It shows the ability of our multi-level scheduling framework to schedule jobs in such a way it maximizes the number of jobs that respect their QoS requirements.
- Percentage of system and software licence usage. It shows as the job classification and scheduling solutions adopted both at MS and LS level allow a fruitful exploitation of the available processors and software licences.
- Average *Slowdown* of jobs without deadline. It shows how the system load delays the execution of such jobs. It is computed as:

$$Slowdown(i) = \frac{Tw_i + Te_i}{Te_i}$$
$$AverageSlowdown = \frac{\sum_i Slowdown(i)}{\#\ of\ processed\ jobs}$$

  Where Tw_i and Te_i are the waiting and the execution time of the job i, respectively. $AverageSlowdown$ closes to 1, it means the scheduler is able to exploit the available computational power avoiding the LS job queues increasing.

In our evaluation we combine the MS with the Flexible Backfilling we developed, and with a variant of the Convergent Scheduler we present in section 3.2.2. In the next sections we will describe the conducted experiments. To evaluate the interaction of MS with the Flexible Backfilling we employ the simulator defined in (43), to evaluate the interaction of MS

with the CS we employ the GridSim simulator (20). In both cases we synthetically generate the streams of jobs we used, this because of the most real streams of jobs we found in literature do not allow the job request of licences and the job deadline time specification (25; 36). Furthermore, the real streams of jobs we analyzed concern with the execution of jobs on a homogeneous cluster, instead, we design our platform in such a way it is composed of different machines, where each one can be a cluster, or a workstation, or a multicore machine, spread on different sites.

4.3.1 Meta-Scheduler - Flexible Backfilling Interaction

The objectives of this experimental phase is to evaluate the feasibility of the Meta-Scheduler scheduling policies, and of the job classification we proposed (Section 3.1). Three different cases were evaluated:

1. MS Heuristics: in which, at LS level scheduling decision are made by means of a Flexible Backfilling algorithm, which exploits job priorities computed by MS. Any job classification is performed at LS level. Higher the job priority is, higher the position of the job in LSs' queues is.

2. LS Heuristics: in which, at LS level we employed the Flexible Backfilling presented in section 3.2.1. The incoming jobs are classified by exploiting the heuristics defined for our FB[3], and jobs are scheduled on the cluster machines by using a Flexible Backfilling algorithm. Job priorities are recomputed at each new scheduling event (submission/ending of a job). This introduce a computational cost not present in the previous case.

3. NO Heuristics: in which jobs are scheduled at both MS and LS level according to the FCFS order without computing priorities. At LS an EASY Backfilling algorithm is used.

[3]Heuristics constant values used in our tests: Deadline ($MAX = 20.0, MIN = 0.1, k = 2$), Anti-aging ($age\_factor = 0.01$), Licences ($W = 1$), Wait minimization ($priority\_boost\_value = 2.0$).

Table 6: Meta-Scheduler - Flexible Backfilling Interaction: Range of values used to generate streams of jobs, machines and software licences.

Parameters	Jobs	Machines	Sw Licences
Estimated	8000-10000		
Benchmark	100-500	100-500	
Margin	1500-5500		
CPU	1-8	4-32	
Licence needs	30%		
Licence ratio			50%-70%
Licence Suitability			[100%]

The evaluation of the interaction between MS and the Flexible Backfilling we propose was conducted by using the described event-based simulator (43), exploiting four streams of jobs with jobs inter-arrival times (Ta in Figure 25) fixed equal to 0, 5, 10, and 15 simulator time unit.

The job submission time is driven by the $wall\_clock$. When the $wall\_clock$ reaches the job submission time, the job enters in the simulation. The simulation ends when all jobs are elaborated. We use the space sharing policy to allocate jobs on machines.

The evaluation was conducted by using a stream of 5000 not preemptable, independent jobs, 20 different types of software licence, and a grid composed by 225 machines, distributed on four different clusters, each one including 120, 60, 30, and 15, respectively.

For each simulation, we randomly generated streams of jobs, whose parameters are generated according to a uniform distribution in the ranges of values showed in Table 6. Both job, machine, and licence parameters are generated according to the range values showed in Table 6.

In order to obtain stable values, each simulation was repeated 20 times with different job attributes values.

For each simulation, 30% of jobs were generated without deadline, and the deadlines of the job i is generated according to the following expression:

$$deadline_i = Submission_i + Estimated_i + Margin_i$$

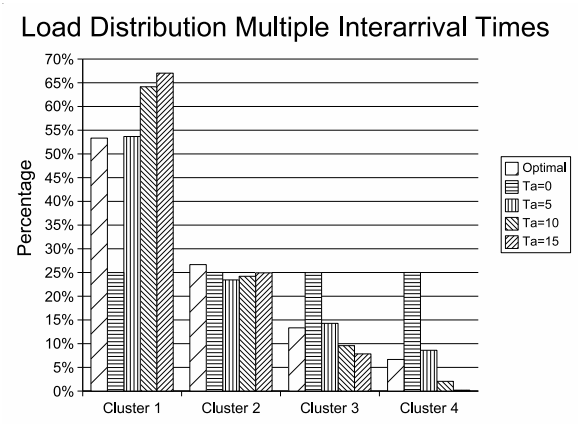

Figure 25: Average clusters load.

Figure 25 shows the average percentage of workload assigned to each cluster through simulations. Such percentage is computed as the ratio between the workload due to the jobs assigned to a cluster, and the workload due to all the jobs in a simulation.

The workload distributions carried out by the proposed policy are functions of the job stream to be scheduled, the job inter-arrival times, and the clusters computational power. Since at LS level a Backfilling algorithm is used, the exploitation of cluster machines is function of the number and kind of jobs queued to a cluster. Greater the number of jobs of different kind is, greater the capacity of Backfilling to efficiently exploit the underlying computational resources is. A more efficient exploitation of the machines of a cluster, leads to quickly "unload" its LS queue. This way improves the probability that such cluster will receive more workload than the less efficiently used ones.

The optimal cluster workload distribution ($Optimal$ in Figure 25) is computed as the ratio between the number of machines belonging to a cluster and the number of available machines. $Ta = 0$ simulates the case in which all jobs are submitted at the same time, and are dispatched before to start their execution. Since MS dispatchs jobs according to the

workload due to LS queued jobs, all clusters obtain the same amount of workload.

The percentage of workload elaborated by each cluster changes according to Ta, and the clusters computational power. Increasing Ta, it could happen that some clusters are enough powerful to maintains empty or "unloaded" their LS queue, with respect to other cluster queues. Consequently, MS dispatches a larger number of jobs to such clusters. This is shown moving from $Ta = 5$ to $Ta = 15$. $Ta = 5$ obtains a workload distribution that better approximates the optimal one. It is because of the amount of workload due to the LS queued jobs, properly represents the clusters computational power. It means that the proposed policy is able to dispatch jobs among underlying clusters, distributing the workload proportionally to the actual cluster computational power. To figure out the quality of the MSs job classification, we show the results obtained by using $Ta = 5$ concerning the other studied metrics.

Figure 26 shows the average Slowdown evaluated for each cluster considering jobs without deadline. It improves in inverse proportion to the cluster computational power. This because of MS scheduling behavior. MS dispatches jobs among clusters with respect to their queued jobs, and its objective is to maintain the same workload in each cluster queue. Consequently, the average jobs slowdown grows to the decreasing computational power of each cluster.

Figure 27 shows the percentage of jobs executed and not respecting their deadlines. It can be seen that the solution based on the LS and MS heuristics are able to improve the number of jobs executed within their deadline, compared to the EASY Backfilling solution. Furthermore, it can be seen that using only MS job classification carries out results comparable to those obtained by LS, with a smaller computational cost. Furthermore, we have to consider the job distribution performed by MS. This implies that the number of jobs executed without respecting their deadline in the case of the cluster 1 it means hundred of jobs, instead in case of the cluster 4 means only few jobs.

Figures 28 and 29 show the percentage of system and software licence usage, respectively. These values are computed according to the follow-

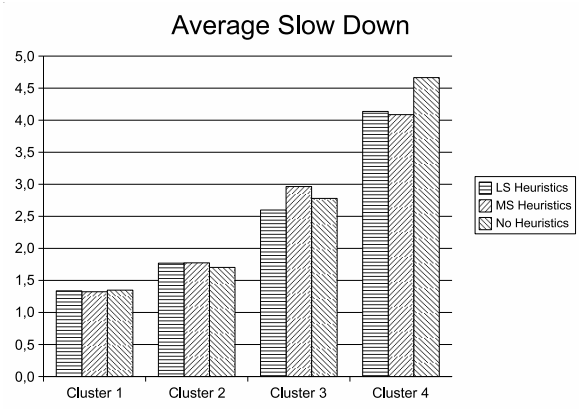

Figure 26: Average Slowdown of jobs without deadline.

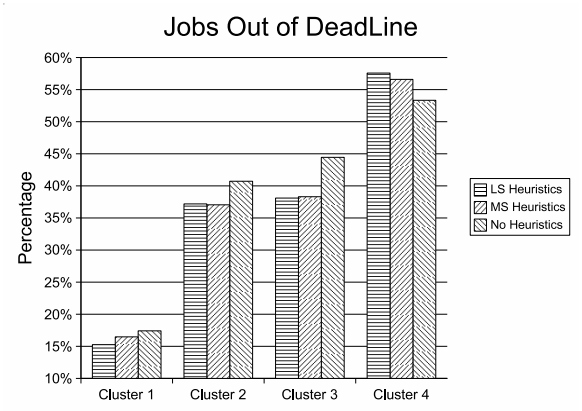

Figure 27: Percentage of jobs executed missing their deadline.

ing expression:

$$\frac{\#\ of\ active\ res}{min(\#\ of\ available\ res, \#\ of\ res\ requested\ by\ jobs)}$$

where res means "processors" or "software licences" when system or software licence usage is computed, respectively.

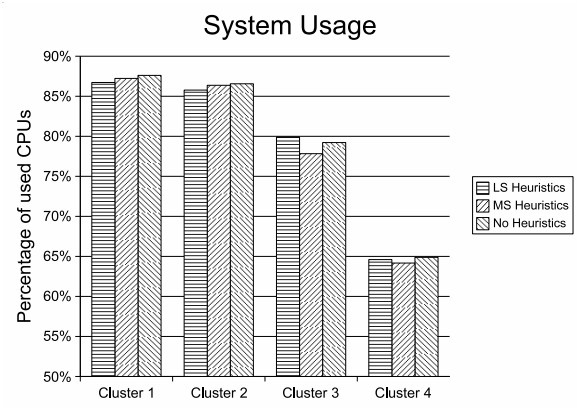

Figure 28: Average percentage of processor usage.

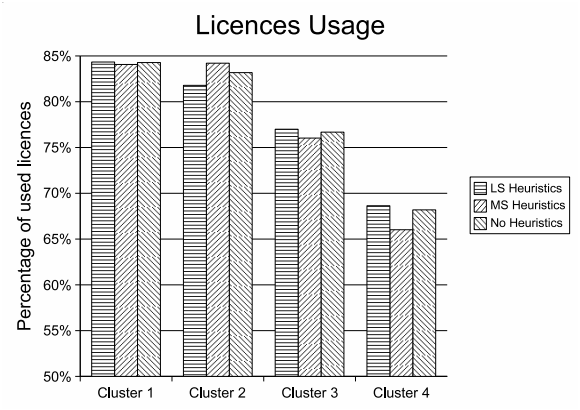

Figure 29: Average percentage of sw licence usage.

All solutions obtain similar results. Smaller is the number of the machines within a cluster, greater are both the processor fragmentation and the contention on software licences, and it leads to a smaller system and software licence usage.

4.3.2 Meta-Scheduler - Convergent Scheduler Interaction

The interaction of MS with the Convergent Scheduler was conducted exploiting the GridSim simulator (20).

The GridSim toolkit provides a facility for simulation of different classes of heterogeneous resources, users, applications, resource brokers, and schedulers. It can be used to simulate application schedulers for single or multiple administrative domain distributed computing systems such as clusters and Grids. Application schedulers in the Grid environment, called resource brokers, perform resource discovery, selection, and aggregation of a diverse set of distributed resources for an individual user. This means that each user has his or her own private resource broker and hence it can be targeted to optimize for the requirements and objectives of its owner. In contrast, schedulers, managing resources such as clusters in a single administrative domain, have complete control over the policy used for allocation of resources. This means that all users need to submit their jobs to the central scheduler, which can be targeted to perform global optimization such as higher system utilization and overall user satisfaction depending on resource allocation policy or optimize for high priority users. We extend the GridSim simulator in order to exploit it with our scheduling infrastructure.

We combined MS with an extend version of our Convergent Scheduler (43), which support the Space Sharing policy. The evaluation was conducted comparing the results obtained by the Convergent Scheduler with: the FCFS, the EASY Backfilling, and the Flexible Backfilling we proposed. The priority computed by MS are used only by the MS's Ordering scheduling function, at LS level job priorities are computed at each scheduling event only by the CS[4], and Flexible Backfilling exploiting the described heuristics[5] (Section 3.2).

[4]Heuristics constant values used in our tests: Deadline ($MAX = 10.0, MIN = 0.1, k = 1.3$), Input ($IN = 3.0$), Overhead Minimization ($\Delta = 800.0$), Anti-aging ($age\_factor = 0.02$), Licences ($LI = 2.5$), Wait minimization ($W = 4.0$).

[5]Heuristics constant values used in our tests: Deadline ($MAX = 20.0, MIN = 0.1, k = 2$), Anti-aging ($age\_factor = 1.1$), Licences ($W = 1$), Wait minimization ($priority\_boost\_value = 2.0$).

Table 7: Meta-Scheduler - Convergent Scheduler Interaction: Range of values used to generate streams of jobs, machines and software licences.

Parameters	Jobs	Machines	Sw Licences
Estimated	8000-10000		
Benchmark	100-500	100-500	
Margin	1500-5500		
CPU	1-8	4-32	
Licence needs	30%		
Licence ratio			50%-70%
Licence Suitability			[100%]

We used a stream of 5000 not preemptable, independent jobs, 20 different types of software licence, and a grid composed by 225 machines, distributed on four different clusters, each one including 120, 60, 30, and 15, respectively. We employed four streams of jobs with jobs inter-arrival times fixed equal to 0, 5, 10, and 15 simulator time unit. Both job and machine parameters are generated according to the range values showed in Table 7.

Figure 30 shows the average percentage of workload assigned to each cluster through simulations. Such percentage is computed as the ratio between the workload due to the jobs assigned to a cluster, and the workload due to all the jobs in a simulation. The optimal cluster workload distribution ($Optimal$ in Figure 30) is computed in the same way as for the MS - Flexible Backfilling interaction: the ratio between the number of machines belonging to a cluster and the number of available machines.

In Figure 30, $Ta = 0$ simulates the case in which all jobs are submitted at the same time, and are dispatched before to start their execution. As described in the previous section, the goal of MS is to balance the workload among LS using information about jobs stored in each LS queue. This means that, if no jobs are executed at LS level, MS assign to each cluster the same amount of workload.

Increasing Ta some clusters are enough powerful to maintains empty or "unloaded" their LS queue with respect to other cluster queues. In

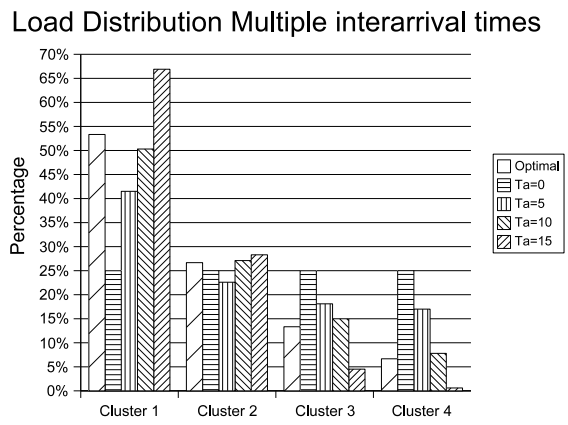

Figure 30: Average clusters load.

this case, MS is able to estimate the computational power of each cluster, and it assigns jobs according to both the stream inter-arrival time feature, and the cluster computational power. Moving from $Ta = 5$ to $Ta = 15$ it can be seen that $Ta = 10$ obtains a workload distribution that better approximates the optimal one. It is because of the job stream features allow MS to fruitful exploit the underlying computational platform. To figure out the quality of the LS scheduling, we show the results obtained by using $Ta = 5$ concerning the other studied metrics. $Ta = 5$ shows that the jobs wait for execution for a long time. This implies that MS assigns a percentage lower than the optimal one to the most powerful cluster and a percentage higher than the optimal one to the less powerful cluster. In this way, the jobs assigned to the less powerful cluster are disadvantaged respect to the others. $Ta = 15$ shows a scenario in which the job stream is not sufficient to exploit all the computational power of the simulated platform. The inter-arrival time is too large, and the most powerful cluster is able and sufficient to execute the almost the 70% of the jobs. In this case, the percentage of jobs assigned to the smallest cluster is close to 0.

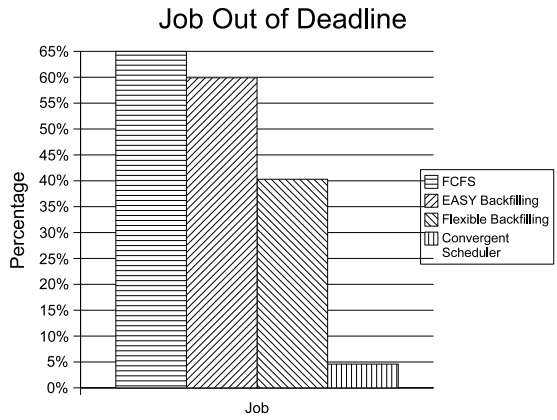

Figure 31: Percentage of jobs executed missing their deadline.

In figure 31 it is shown the percentage of jobs executed without respecting their deadline using different algorithms as Local-Scheduler. The aim of the Convergent Scheduler is to maximize the number of job executed whiting their deadline, and figure 31 shows that CS outperforms the other exploited algorithms, scheduling jobs in such a way that almost the 97% execute whiting their deadline. The percentage of jobs executing out of their deadline increases exploiting the Flexible Backfilling, because, even if it addresses the deadline constraint, it does not exploit dynamic resource information to make scheduling decisions (Section 3.2.1), the EASY Backfilling and the FCFS, because these algorithms don't address explicitly the deadline constraint.

In figure 32 it is shown the average Slowdown for the jobs without deadline. In this case the performance of the Convergent Scheduler is worse than the other algorithms because it delays the execution of such jobs to favor the execution of the jobs with deadline.

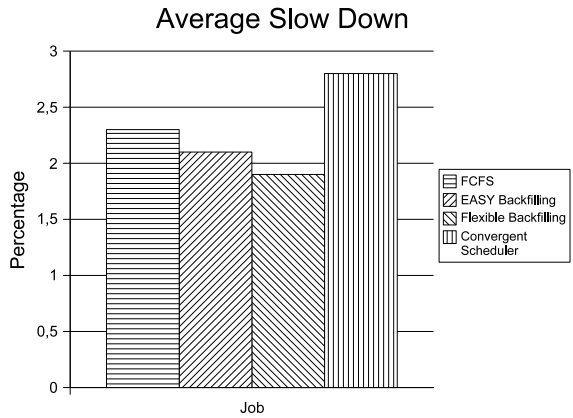

Figure 32: Average Slowdown of jobs without deadline.

Chapter 5

Related Results

In this chapter we present two research lines we follow during the Thesis study, which are not part of the main topic of the Thesis.

The first ones concerns to the application of autonomic aspect to the Meta-Scheduler classifier. We focus our research on the self-optimizing aspect (Section 5.1). We propose a formal definition of a self-optimizing classification system, and a design pattern exploitable to implement such kind of systems (30). Eventually, we show the feasibility of our methodology, and we present some preliminary experiments.

The second research line concerns to the comparison of our Flexible Backfilling with a *schedule-based* system proposed by Dalibor Klusáček and Hana Rudová of the University of Brno (62). These kind of schedulers make resource planning for the present and the future, which results in an assignment of start times to all jobs on a specific machine. This results in a different approach to the scheduling problem from that presented in the Chapter 3, which is a *queue-based* scheduler. We present the solution developed by Klusáček based on *Earliest Gap-Earliest Deadline First*, and *Tabu Search* techniques. Eventually, we show the results carried out by the comparison of our Flexible Backfilling and the proposed schedule-based system.

5.1 Autonomic Aspects

The classification mechanism we presented in the Chapter 3 exploited by the Meta-Scheduler is statically defined. This means that it exploits a set of heuristics applying them to classify incoming jobs without considering any dynamic changes of the job characteristics. In such a way, it can happens that the carried out classification does not "expresses and aggregates" the job QoS requirements, and two distinct jobs, with different QoS requirements, could be labeled with the same priority value. To overcome this limitation we studied the features of autonomic systems, and how to apply them to our classifier (12; 78; 79).

The *Autonomic Computing* is an initiative started by IBM in 2001 (aka ACI (55)), with the presentation of its manifesto (56). Its aim is to create self-managing computer systems to overcome their rapidly growing complexity and to enable their further growth. The manifesto states that a system, to be autonomic, must have the following properties:

- self-configuring, which is the ability of systems to handle reconfiguration inside thyself,
- self-healing, which is the ability of systems to provide their services in spite of failures,
- self-optimizing, which is the ability of systems to adapt their configuration and structure in order to achieve the best/required performance.
- self-protecting, which is the ability of systems to predict, prevent, detect and identify attacks, and to protect itself against them

In this section we describe which are the fundamental elements needed to make self-optimizing a strategy-based job classification system. The aim is to provide to the Meta-Scheduler mechanisms to recognize mismatches between actual and expected classifier behavior (with respect to a target behavior), and to provide to it the capability to adapt itself.

Our approach is not intended to be either too abstract or too formal. We describe with a high-level notation the elements that a designer needs to care about when designing a self-optimizing classifier. The challenge is to provide technique that enables software systems to evolve in order to remain useful (5), but to do so in a way that does not incur downtime as traditional maintenance processes do (91).

5.1.1 Formalization of Priority Classification Systems

A priority classification system S can be formally described with the following higher-order function:

$$f_S \quad : \quad I \times F_{strategy} \longrightarrow O. \tag{5.1}$$

Where I is the set of all possible item that need to be classified, $F_{strategy}$ is the set of all the possible strategy-functions, and $O \subset \mathbb{N}$ is a finite set of output values. A strategy-function $f_{strategy} \in F_{strategy}$ is defined as:

$$f_{strategy} : I \rightarrow O$$

The system S applies $f_{strategy}$ to each input $i \in I$ in order to obtain a priority value $p \in O$.

A priority classifier system can be defined self-optimizing when it is able to adapt itself in order to maintain good classification performance whatever an input comes into the system. From our perspective, a priority classifier offers a good performance when it is able to satisfy two requirements:

- the priority value assigned to each input item is well-proportioned to the item relevance (i.e. very important item must have a very high priority)
- the priorities assigned by the system must be coherent with a specified target policy

There are two different approaches to design a classifier able to achieve good performance, hence address the reported requirements. The first one consists in using a $f_{strategy}$ designed with a deep knowledge about the items that the classifier has to classify: a strategy-function strictly tailored to the item that will be classified, able to manage every possible input stream. The other approach is to design a self-optimizing classifier able to change the strategy-function taking care of the priorities assigned to a finite portion of the past classified items. This partial information is used by the self-optimizing classifier to analyze the trend of the distribution of the priorities. Such information drives the tuning of the strategy-function behavior.

To model the self-optimization nature of the classifier we need to enrich the definition previously introduced for the strategy-function based classification system. Namely, we need to define how the *strategy function* can be modified to enhance its performance. This requires a *reconfiguration mechanism* able to modify the strategy function and an *evaluation mechanism* able to evaluate the *historical data* and to drive, through the reconfiguration mechanism, the changes in the strategy-function. As we will show in section 5.1.3, we employ this key concepts to define a design pattern for autonomic stream-classification-systems.

Formally, the reconfiguration and evaluation mechanisms can be modeled using the two following functions:

$$F_{eval} \quad : \quad H \times C \longrightarrow C \tag{5.2}$$

$$F_{reconf} \quad : \quad C \longrightarrow F_{strategy} \tag{5.3}$$

Where H is the set of all possible historical data. C is the configuration used by F_{reconf} to select an appropriated strategy-function among the available.

Every time the system completes the computation of a new item, F_{eval} evaluates the priority distribution $d_{current}$ obtained analyzing the current historical data $h_{current} \in H$. If the result of this evaluation highlight an incoherence between the $d_{current}$ distribution and the target policy d_{target}, F_{eval} generates a new configuration $c \in C$. If F_{eval} does not recognize any

incongruence, it simply returns as new configuration the current one.

When a new configuration is available, F_{reconf} uses this configuration in order to select a novel and more appropriate $f_{strategy}$ in the $F_{strategy}$ set.

5.1.2 Polytope

In the formalization proposed, the self-optimizing classification system can access to an unlimited set of historical data, and it generates a new strategy function if F_{eval} recognized any difference between the $d_{current}$ and d_{target}. However, more realistic systems can only access to a finite set of historical data, and their reconfiguration mechanisms do not generate a new strategy function every time that the classifier does not work as expected. Indeed, in a real scenario, the evaluation mechanism should trigger strategy function changes only when the current behavior of classifier is quite different to the expected one.

In order to formalize these concepts, we introduced the concept of polytope. Consider the output of F_{eval} as a point in a geometric space. The value range in which it is free to move, without implying a reconfiguration of the strategy-function, is a *polytope*.

We can define the polytope $\mathcal{P}$ of a strategy-function S the set of F_{eval} outputs c that don't trigger the strategy-function reconfiguration. Formally:

$$\mathcal{P} = \{S_c \mid Acceptance_S(S_c)\}$$

where $Acceptance_S$ is a boolean function that returns true if the strategy-function behavior is acceptable.

From a formal point of view, in order to consider the polytope, the F_{eval} function must be changed. Indeed, it has to generate a new $f_{strategy}$ configuration only if $d_{current}$ does not belong to $\mathcal{P}$:

$$F_{eval} : H \times C \times \mathcal{P} \longrightarrow C$$

5.1.3 Self-Optimizing Design Pattern for Priority Classification Systems

In this section, we present our self-optimizing behavioral design pattern derived from the formalization of a strategy-driven classification system. The aim is to provide a general repeatable solution easing the design of self-optimizing classification systems.

According to our formalization, a classification system is characterized by three components: an InputStream, a Classifier and an OutputStream. They can be represented in the following way:

- INPUTSTREAM: a stream (or set) of independent elements I among which there are not functional dependencies.
- CLASSIFIER: a classification function (f) applied to each element of I.
- OUTPUTSTREAM: a stream (or set) of elements O such that each element is $e'_i = f(e_i)$ with $e_i \in I$ and $e'_i \in O$.

The CLASSIFIER retrieves each input element from the INPUTSTREAM, then it classifies the element eventually sent to the OUTPUTSTREAM. Typically, the classification is driven by a policy specified by the classifier administrator.

Conceiving our self-optimizing pattern for classification systems we have taken inspiration from the GoF strategy pattern (44). The main entity of our pattern is called STRATEGY. It is able to classify the items, to evaluate itself, and to change its behavior accordingly to some rules. To perform these tasks STRATEGY uses three entities: DATAREPOSITORY, EVALUATOR and RECONFIGURATOR. Their behavior can be described as follows:

- DATAREPOSITORY: it is an entity that holds up to a (finite) number of past input elements coupled with the respective computed outputs.
- EVALUATOR: it is an entity able to suggest a STRATEGY reconfiguration. It takes as input the current configuration of the STRATEGY

and using the knowledge stored in the DATAREPOSITORY, EVALUATOR suggests a change in the STRATEGY behavior.

- RECONFIGURATOR: it is an entity that, taking as input the STRATEGY and the output of EVALUATOR, it is able to reconfigure the STRATEGY, acting onto its specific tuning parameters, in order to optimize its performance.

The CLASSIFIER forwards the items retrieved from the INPUTSTREAM directly to the STRATEGY. Before classifying the items, The STRATEGY evaluates its own configuration by invoking the EVALUATOR. The EVALUATOR reads the past input/output from the DATAREPOSITORY and then it evaluates the adherence of the classifier behavior with respect to the given classification policy. If the behavior is different from the expected one, the EVALUATOR suggests a change in the STRATEGY configuration.

If a reconfiguration is needed, STRATEGY invokes RECONFIGURATOR passing it, as input parameters, itself and the reconfiguration suggested by the EVALUATOR. The RECONFIGURATOR retrieves the tuning parameters of the STRATEGY through which it changes the configuration and, in consequence, the behavior of STRATEGY. After the reconfiguration step the STRATEGY computes the output values accordingly to its new configuration and stores both the input and the computed output into DATAREPOSITORY. Finally, the STRATEGY sends the computed output back to the CLASSIFIER that in turn send it to the OUTPUTSTREAM.

A UML schema of the packages and classes that implements our self-optimizing design pattern is depicted in Figure 33. The higher part of the figure represents the classification system, made up of the CLASSIFIER entity, the INPUTSTREAM entity and the OUTPUTSTREAM entity. In the lower part of the figure, it is represented our self-optimizing pattern belonging to a package. The pattern package is made up of four entities: the EVALUATOR, the STRATEGY, the DATAREPOSITORY and the RECONFIGURATOR.

In Table 8 we point out how the formal model of we defined for a classifier can be mapped into the proposed design pattern.

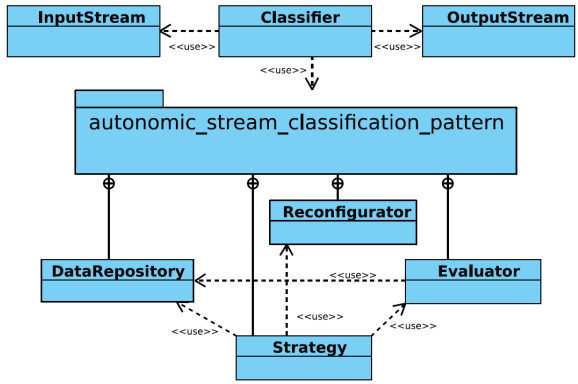

Figure 33: UML schema of the proposed autonomic strategy pattern.

Abstract Model Functions	Design Pattern Components
I	InputStream
$f_{strategy}$	Strategy
F_{eval}	Evaluator
F_{reconf}	Reconfigurator
H	DataRepository
O	OutputStream

Table 8: Abstract Model - Design Pattern mapping

5.1.4 Case study

In this section we describe how to exploit the proposed model applying the design pattern we defined to a real scenario. We developed a classifier derived from that used by the Meta-Scheduler. In particular, we design a self-optimizing classifier based on the Deadline heuristics defined in section 3.1.1. This choice was driven by our will to implement a simple case of study, in which operates only one heuristics. In this way, it is simple to recognize bad heuristics behaviors, and, it also makes easier to understand how the self-optimizing classifier can change its behavior. In section 5.1.5, we present some preliminary results, and we show the feasibility of our design pattern.

To make the Deadline heuristics self-optimizing, and to enable it to compute a profitable priority distribution in any situation, we need to point out the cause of unprofitable priority distributions: the large differences of the $Margin$ value among jobs (Section 3.1.1).

For instance, suppose that the job stream could be divided in two sub-streams: in the first one the job are characterized by high values of the $Margin$ parameter, in the second one by low values.

In the boundary regions between the two consecutive job sub-streams, Deadline produces a classification in which jobs priorities are concentrated in extremely low (or high) values. Nevertheless, the Deadline heuristics continues to perform properly if each job sub-stream length is sizable. This is because of after a transient state the average $Margin$ used by the heuristics changes approaching to the average $Margin$ of the current job sub-stream. It permits to the Deadline heuristics to work properly, i.e. to compute job priorities with a distribution faithful to the exponential one that we refer as the optimal.

To avoid this situation, we should be able to change the way in which Deadline assigns priorities, and to maintain a proper priority distribution when the average $Margin$ changes rapidly.

In our implementation of a self-optimizing classifier, we follow the design pattern previously described. We can map our classifier in a general classification systems defined by (5.1), where I is the input job stream, the Deadline heuristics is our strategy function $f_{strategy} \in F_{strategy}$, and O is the outgoing jobs labeled with a priority value.

The abstract model defines a set of values for the classifier parameters, in such a way that is possible to modify the classifier behavior changing those values. In our implementation, we enrich Deadline with a parameter $base$, in such a way it is able to change its behavior. We modify the expression (3.4) in:

$$MinUnity = \frac{2 * E[Margin]}{\sum_{k=1}^{max} base^k}$$

In this way the interval subdivision showed in figure 10 can be changed dynamically changing the $base$ parameter.

We designed the $Eval$ and $Reconf$ components, which implement the F_{eval} (5.2), F_{reconf} (5.3) functions respectively. The historical data (i.e. $Data$ component) and the configuration setting taken as input by Eval are the priority values of the past analyzed jobs, and the current $base$ value respectively. The output of the Eval component is a new value of the $base$ parameter. Reconf produces a new $f_{startegy}$ in which the $base$ parameter is changed according to the Eval output.

In Table 9 we point out how the design pattern components can be mapped into our classifier implementation.

Classifier components	**Design Pattern Components**
job input stream	InputStream
Deadline	Strategy
Eval	Evaluator
Reconf	Reconfigurator
Data	DataRepository
prioritized jobs	OutputStream

Table 9: Classifier Components - Design Pattern mapping

In our implementation, the polytope is a threshold value δ used to accept, or not, the reconfiguration carried out by the RECONFIGURATOR.

In order to point out how polytope is used, we introduce the vector space D with dimension P, where P is the number of possible priority values, which is a way to represent priority distribution of historical data. In particular, each priority distribution, obtained from $h \in H$, is represented with a vector $d = (d_1, ..d_k, .., d_P)$.

In this context, each component d_k of d is a value grater than or equal to zero and smaller than or equal to one ($0 \leq d_k \leq 1 \; \forall_{k=1,..,P}$). It represents the percentage of jobs belongings to historical data h to which the system has assigned the priority $k \in P$. Since each component d_k is a percentage grater than zero and smaller than one, the sum of all d_k must

be equal to one ($\sum_{k=1}^{P} d_k = 1$). We enforce this last constraints by using the Norm ($\|.\|_1$).

The polytope is defined as:

$$\mathcal{P} = \{d : \|d - d_{target}\|_1 \leq \delta\}$$

It characterizes the permissible working region as the set of all priority distributions that are "far" from the target distribution of a quantity less than or equal to a certain radius δ. In this context the measure of the distance is performed using the Euclidean Norm ($\|.\|_2$).

In other words, if $d_{current}$ belongs to the polytope $\mathcal{P}$ it means that the distance between $d_{current}$ and d_{target} is smaller than the fixed radius δ, hence $f_{strategy}$ does not need to be reconfigured, otherwise a new reconfiguration is performed.

5.1.5 Experiments

To evaluate the goodness of the *self-Optimizing DeadLine Heuristics* (ADH) solution, we conducted simulations applying the classification algorithm to a stream of jobs characterized by a high variability of the $Margin$ parameter.

To conduct the evaluation, we developed an ad-hoc event-driven simulator. For each simulation, we randomly generated a job-stream whose $Margin$ parameter was generated according to different distributions and described in each test. A simulation step includes: (1) selection and classification of new job, (2) update of the system and heuristics state, (3) verify the correct behavior of the system, and eventually, perform the system adaptation. The time of job submission is driven by the *wall_clock*. When the *wall_clock* reaches the job submission time, the job enters in the simulation.

The aim of the experimentation phase was to carry out a job priority distribution according to an administrative policy constituting the input of the proposed heuristics: system administrators can define a relation among the number and the kind of jobs in the system.

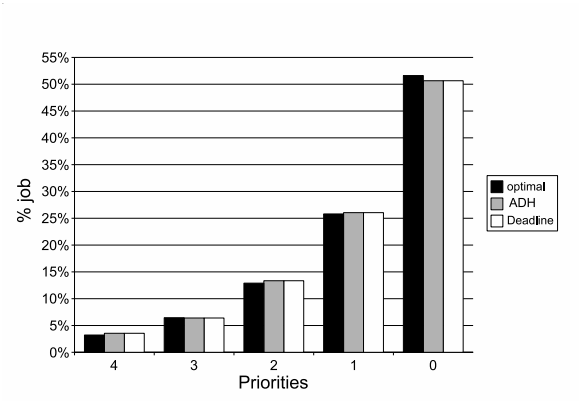

Figure 34: Deadline and ADH evaluation in the case of a uniform distribution of the $Margin$ parameter

In our tests, the relation on the number of jobs for each priority is characterized as relation 3.2. We have five classes of priority in our simulations, afterwards the possible priority values that a job can assume are in $[0, 4]$.

In our experiments we compare the ADH and the DeadLine heuristics (Section 3.1.1). The performance metric used is the faithful of the classification given by these two heuristics, with the classification given by the exponential distribution described in 3.2, which we consider the optimal one in our case study.

Figure 34 shows the behavior of the DeadLine heuristics, and of its self-optimizing version (ADH), compared to the optimal solution, when the $Margin$ of each job is uniformly generated. In this case ADH is never invoked because the DeadLine heuristics is good enough to model this situation and there is not need to operate to modify its behavior.

Table 10 shows the range values used to generate the $Margin$ in the uniform case. The first column of the table shows sub-streams of jobs that have the same range of $Margin$, the second column shows the range for the specific job sub-stream. Obviously, in the uniform case all the jobs belong to the same range.

#Jobs	Margin Range
2000	0-200

Table 10: Range values for the $Margin$ parameter in the case of uniform distribution

Figure 35 shows the trend of the $Margin$ parameter in a non-uniformly distributed jobs generation, compared with the behavior of the *base* value (b in figure), that is the base to compute the division of the range in QoS buckets. We can note that the average $Margin$ quickly changes in a not predictable way. Moreover, *base* of the ADH frequently changes in order to control the generated jobs priority distribution. Table 11 shows the four job sub-streams in which we divided the job-stream, and the range of $Margin$ values used in these tests.

#Jobs	Margin Range
400	2000-4000
550	0-200
350	2000-4000
700	100-200

Table 11: Range values for the $Margin$ parameter in the case of a not-uniform distribution.

Figure 36 depicts the priority distribution carried out using ADH and the *DeadLine* heuristics, compared with the optimal solution when the job deadline is generated according to the Table 11. The results point out that the autonomic heuristics lacks of accuracy for the low priority jobs, but it is close to the optimal solution for the jobs with high priority. Furthermore, ADH trend respects the optimal solution. The DeadLine heuristics, by itself, is not able to handle changes in the $Margin$ distribution: this because the $Margin$ parameter falls down too fast with respect to its average value.

The two last figures show the behavior of the system when the $Margin$ parameter is stable for some elements of the job-stream, namely

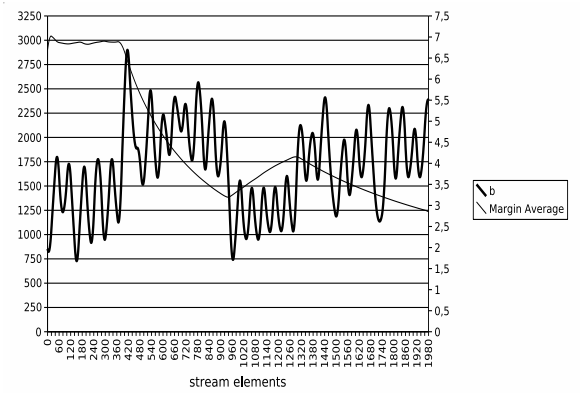

Figure 35: Deadline and ADH evaluation in the case of a not-uniform distribution of the $Margin$ parameter

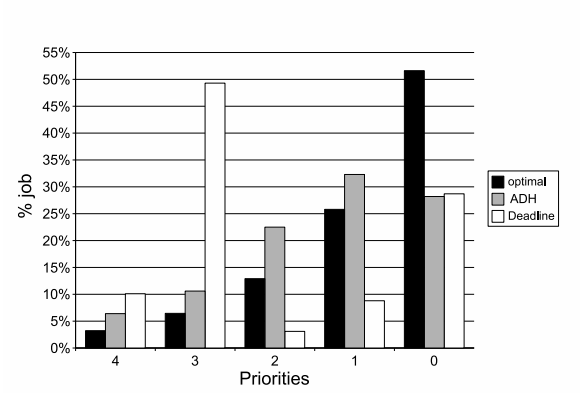

Figure 36: Deadline and ADH evaluation in the case of a not-uniform distribution of the $Margin$ parameter

the intervals [1000-1100] and [1300-1500]. Figure 37 shows that *base* does not change when the slope of the $Margin$ average curve is not steep. This implies that the DeadLine heuristics assignments are profitable. In the other cases, when the curve sheers or falls down, the ADH intervenes to adapt heuristics behavior.

Table 12 shows the job-stream subdivision and the range of $Margin$

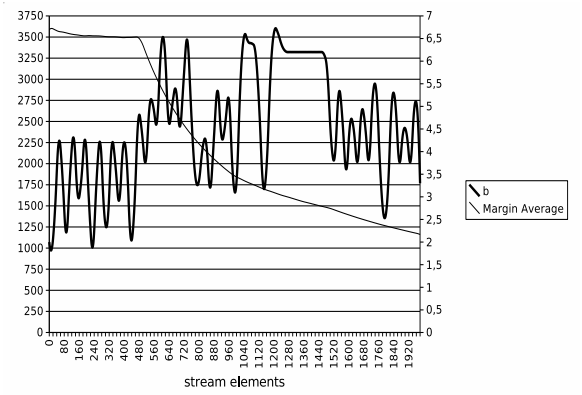

Figure 37: $Margin$ average and *base* trend

values defined for each job sub-stream.

#Jobs	Margin Range
500	3000-4000
500	100-200
500	500-700
500	100-200

Table 12: Range values for the $Margin$ parameter in the case of a not-uniform distribution.

Finally, Figure 38 shows the behavior of the system according to a job deadline generation described in Table 12. The DeadLine heuristics is not able to recognize hot spots in the QoS buckets. In fact, when a lot of jobs with the same $Margin$ are submitted to the system, they receive the same priority. Instead, ADH is able to change the *base* value to compute a better division of the interval and it is able to satisfy the administrator policy.

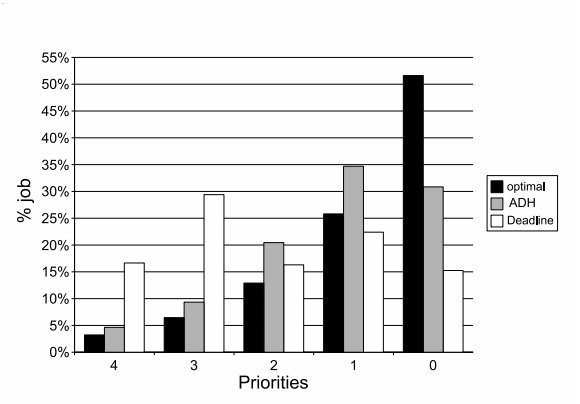

Figure 38: Deadline and ADH evaluation in the case of a not-uniform distribution of the $Margin$ parameter

5.2 Schedule-Based Algorithm

The scheduling framework proposed in Chapter 3 can be classified as a *queue-based* system. A queuing system aims to assign free resources to waiting jobs (60). The highest prioritized job is usually the queue head. If not enough resources are available to schedule any of the queued jobs, the scheduler waits until enough resources become available. As we showed, strategies to improve the quality of the scheduling cab be applied, and a lot of research efforts have been successfully devoted to this purpose.

In this section we present a different approach that was investigated by Klusáček and was compared with our Flexible Backfilling described in section 3.2.1: a *schedule-based* approach (62). This approach allows precise mapping of jobs onto machines in time. This permits to use advanced scheduling algorithms (82), such as local search methods (48), to optimize the computed schedules.

Schedule-based systems make resource planning for the present and the future, which results in an assignment of start times to all jobs on a specific machine. Job execution estimates are mandatory in this approach. With this knowledge, advanced reservations are easily possible. In these systems every incoming jobs are immediately analyzed and they

are planned in the schedule (60). Due to their computational cost, these approaches were mostly applied to static problems assuming that all jobs and resources are known in advance. This allows to create schedule for all jobs at once.

In this section we describe a novel schedule-based solution to schedule dynamically arriving batch jobs on the machines of a computational grid. This approach exploits *dispatching rule*, which is used to create an initial schedule, and *local search*, which optimizes the initial solution according to the objective function, in an incremental fashion (27). This means that current computed schedule can be used as the starting point to build new schedules after each job arrival. This leads to a reasonable computational cost since the schedule is not rebuilt from scratch. A multicriteria approach is proposed, it is based on providing nontrivial QoS to end users, while satisfying the system administrators requirements as well. Moreover, efficient method is developed to detect and fill existing gaps in the schedule with suitable jobs. It allows to increase both the QoS and machine usage by limiting fragmentation of the processor time.

5.2.1 Earliest Gap-Earliest Deadline First Rule

The *Earliest Gap-Earliest Deadline First* (EG-EDF) dispatching rule places a new submitted job into the existing schedule to built the schedule incrementally. In this way, the scheduler can compute new scheduling plans saving running time for scheduling, since a new plan is not re-computed from scratch. To do this, the scheduler has to insert new jobs in the scheduling plan avoiding *gaps*, otherwise, resource utilization may drop quickly. We define a gap as a period in which there are idle CPUs. A new gap appears in the schedule every time some available CPUs can not be assigned to any job because of the number of available processors is lower than the number of requested by each job. In such situation, job has to be placed in the schedule to a time in which a sufficient number of CPUs is available. This introduce a gap in the schedule, and the job must be delayed until there are enough resources needed to perform it. Gaps generally lead to processor fragmentation which results in a bad system

utilization.

In order to reduce the processor fragmentation, a method able to optimize the scheduling plan by placing the jobs into existing gaps is developed. This is a key part of our EG-EDF rule. Suppose a new job i arrives into the system, exploiting the existing schedule, the *Earliest Gap* (EG) suitable to perform the job i is identified for each machine. Let S denotes the number of found EGs suitable to perform i ($S \leq \#\ of\ Machines$) Three different cases are considered: $S \geq 2$, $S = 1$, and $S = 0$.

$S \geq 2$ means there are more than one EG suitable for the job assignment. We compute a *weight* for each schedule carried out supposing to assign the job i to different EG (i.e. to different machines). The schedule with the highest weight is chosen, and i is assigned to the resulting machine EG (i.e. machine). We define the $weight$ function, computed for each assignment of i to the each EG, as follows:

$$
\begin{aligned}
weight &= weight_{(makespan)} + weight_{(deadline)} \\
weight_{(makespan)} &= \frac{makespan_{(old)} - makespan_{(new)}}{makespan_{(old)}} \\
weight_{(deadline)} &= \frac{nondelayed_{(new)} - nondelayed_{(old)}}{nondelayed_{(old)}}
\end{aligned}
$$

Here the $makespan_{(old)}$ is the expected makespan (e.g. the completion time of the last job in the schedule) of the current scheduling plan (i.e. without considering the job i), $makespan_{(new)}$ is the makespan of the new schedule, in which the job i is assigned to one of the found EGs. $nondelayed_{(old)}$ and $nondelayed_{(new)}$ are the number of jobs executed within their deadline before and after the job assignment, respectively.

$S = 1$ simply means there is just one machine with a EG suitable for i, and this is used for the job assignment.

$S = 0$ means that there are no suitable gaps to which assign the job i. In this case, the scheduler computes the $weight$ function supposing to assign i to each machine in the system following the Earliest Deadline

First (EDF) strategy. Each of these assignments is evaluated separately, and the one with the highest weight is accepted.

5.2.2 Tabu Search

EG-EDF allows to include incoming jobs in the scheduling plan optimizing the resource usage and increasing the number of jobs executing whiting their deadlines. EG-EDF does not consider the jobs that are part of the scheduling plan, but it manipulates only with the newly arrived jobs. In such case many gaps in the schedule may remain. To reduce this effect, the Tabu Search (47) optimization algorithm is proposed. It considers only the jobs included in the current scheduling plan and waiting for execution. Tabu Search can be used also considering the running jobs if preemption is allowed.

Tabu Search selects the job foreseen the last to be executed in the scheduling plan of the machine with the highest number of delayed jobs. Such job is added to the *Tabu List* if it is not jet present, otherwise it is discarded to avoid cycling. The Tabu List is built incrementally at each iteration, and it has a limited size. Jobs are added to the Tabu List in such a way that the oldest inserted is removed when the list becomes full.

Tabu Search analyzes the selected job to find the EG suitable to perform it in the current scheduling plan of the machine with the highest number of delayed jobs. If a suitable EG is found, Tabu Search computes the $weight$ function supposing to move the job in this EG scheduling position.

If $weight > 0$ it means that moving the job improves the quality of the current scheduling plan. So, Tabu Search accepts the new scheduling plan (i.e. it moves the job), the $makespan_{(new)}$ and $nondelayed_{(new)}$ values are computed, and Tabu Search can perform a new iteration.

Otherwise, if $weight = 0$ it means that the quality of the current scheduling plan is not improved. In this case, Tabu Search does not move the job, and it searches a new EG suitable to perform the job in the scheduling plan. If none of the remaining machines has a suitable gap in its schedule, a new iteration is started by selecting a different job not

present in the Tabu List. It can happen that the machine with the highest number of delayed jobs contains only jobs present in the Tabu List. Then the machine with the second highest number of delayed jobs is analyzed to select jobs to which will be applied the Tabu Search procedure. The process continues until there are no delayed non-tabu jobs, or the upper bound of iterations is reached.

5.2.3 Experiments

In order to evaluate the feasibility of the EG-EDF and Tabu Search solutions, some experiments have been conducted. The evaluation was performed by comparing this solution with FCFS, EASY backfilling (Easy BF), and our Flexible backfilling presented in section 3.2.1 (Flex. BF).

Concerning the Flex. BF, job priorities are updated at each job submission or ending event, and the reservation for the first queued job is maintained through events.

We employ the Alea Simulator, which is an extended version of the GridSim toolkit (20; 28). The evaluation was conducted by simulations using five different streams of jobs synthetically generated. The jobs submission time is generated according to a negative exponential distribution with different inter-arrival times between jobs.

According to the job inter-arrival times a different workload is generated through a simulation. Smaller this time is, greater the system workload is. Fixing the jobs Inter-arrival time equal to 5 time units allows to avoid the job queue increasing exponentially with respect to the available computational power (i.e. the simulated platform). Moreover, each job and machine parameter was randomly generated according to a uniform distribution in the range showed in table 13, where $machine\ speed$ represents the architecture used both for the $job\ execution\ time$ parameter generation, and the considered machine computational power.

The experiments were conducted by simulate a grid platform made up of 150 machines with different CPU number and speed, and a stream of 3000 jobs (which parameters are generated according to the range in Table 13). Job scheduling plans were carried out by exploiting the Space

Parameters	Range of values
job execution time	$[500 - 3000]$
jobs with deadlines	70%
number of CPUs required by job	$[1 - 8]$
number of CPU per machine	$[1 - 16]$
machine speed	$[200 - 600]$

Table 13: Range values for the job and machine parameters.

Sharing processor allocation policy, and both parallel and sequential jobs were simulated. In order to obtain stable results, each simulation was repeated 20 times with different job streams.

To evaluate the quality of schedules computed by EG-EDF rule and Tabu Search, we exploited different criteria: the percentage of jobs executed missing their deadline, the percentage of system usage, the average job slowdown, and the time due to the computation of the scheduling plans.

The system usage was computed at each simulation time by using the following expression:

$$System\ usage = \frac{\#\ of\ active\ CPUs}{min(\#\ of\ available\ CPUs, \#\ of\ CPUs\ requested\ by\ jobs)}$$

It allows to not consider situations when there are not enough jobs in the system to use all the available machines. It happens at the beginning and at the end of the simulation.

The slowdown is computed as $(Tw + Te)/Te$, with Tw the time that a job spends in queue waiting to start its execution, and Te the job execution time. The slowdown shows how the system load delays the execution of jobs.

In Figure 39 (left) the percentage of jobs executed not respecting their deadline is shown. As expected, when the job inter-arrival time increases, the number of late jobs decreases. Moreover, it can be seen that both EG-EDF rule and Tabu Search produced much better solutions than Flexible

Backfilling, Easy Backfilling, and FCFS. Tabu Search outperforms all the other algorithms. In particular, it obtains the same results of EG-EDF rule when the system contention is low (job inter-arrival time equal to 5). In Figure 39 (right), the percentage of system usage is shown. Schedule-based algorithms are, in general, able to better exploit the system computational resources. However, when there is not contention in the system the solutions we propose obtained worse results than the other ones. When the available computational power is able to avoid the job queue increasing, the Tabu Search and EG-EDF solutions do not improve, or improve very little, the previous schedule. In this situation, the schedule-based approach is less effective concerning the resource utilization. In such situation the schedule is almost empty so a newly arrived job is often immediately executed on an available machine, therefore the Tabu Search has a very limited space for optimization moves.

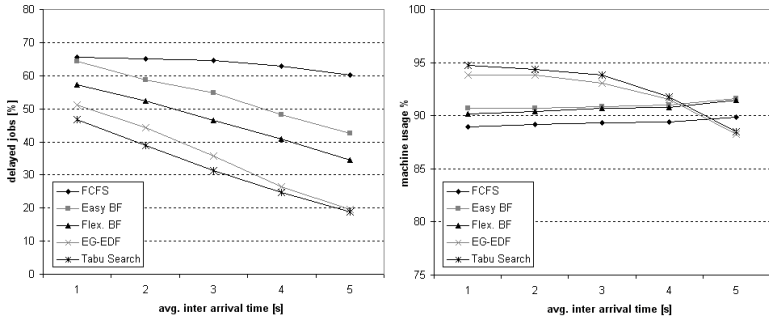

Figure 39: Average percentage of delayed jobs (left) and system usage (right).

Figure 40 (left) shows the average scheduling times spent by the scheduler for conducting the tests on the simulated computational environment. It is computed by measuring the scheduling time at each scheduling event. The runtime of FCFS is very low w.r.t. to Easy and Flexible Backfilling for which it grows quickly as a function of the job queue length. Although the Flexible Backfilling has to re-compute job priorities at each scheduling event, and then has to sort the queue accordingly, it

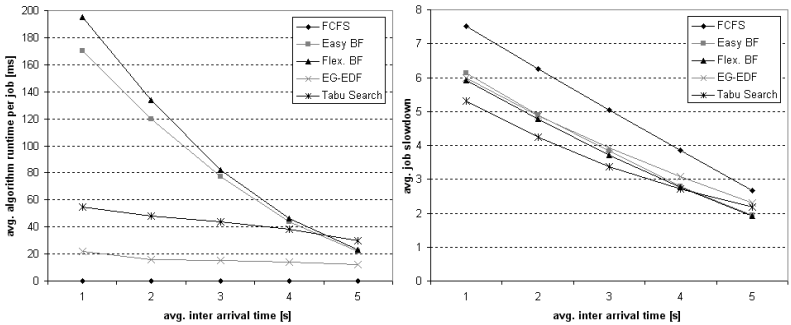

Figure 40: Average algorithm runtime (left) and the average job slowdown (right).

causes minimal growth of its run time compared to the Easy Backfilling. This is due to the application of an efficient sort algorithm.

Local search based algorithms are often considered to be very time consuming. Our implementation, which exploits an incremental approach based on the previously computed schedule, is able to guarantee a shorter and stable execution time w.r.t. the other algorithms. In particular, EG-EDF rule is fast and it always generates acceptable schedule, so we can stop Tabu Search optimization at any time if prompt decisions are required.

Figure 40 (right) shows the average job slowdown. This shows us how the system load delays the job execution. As expected, greater the system contention is, greater the job slowdown is. In this case the better results are obtained by Tabu Search, which are enough close to those obtained by the Flexible Backfilling algorithm.

Chapter 6

Conclusion and Future Works

Grids enable the sharing, selection, and aggregation of a wide variety of different resources (e.g supercomputers, storage systems, data sources, and specialized devices), for solving large-scale computational and data intensive problems in science, engineering, and commerce. The resources in grids are heterogeneous and geographically distributed, and are located in different administrative domains. This lead to create virtual organizations, as a temporary alliance of enterprises and/or organizations, which common aim is to share resources, skills, and competencies, in order to better respond to business opportunities, or to scientific and engineering challenges.

To build a grid infrastructure, it is required that a number of services are developed and deployed. From the grid administrators' point of view, this includes security and information services, accounting mechanisms, resource aggregation and allocation. From the users' point of view, services for application development, scheduling, and execution management are needed.

In a grid environment the scheduler plays an important role when issues like acceptance, usability, or performance of applications and resources are considered. Grid schedulers assign jobs to specific time

intervals of resources, such that no two jobs are on any resource at the same time, or such that the capacity of the resource is not exceeded by the jobs. The computed job schedule is optimal if it minimizes/maximizes a given optimality criteria.

In this Ph.D Thesis we studied the grid scheduling problem in one of its restricted form. We designed a hierarchical grid scheduler able to manage a set of distributed and heterogeneous resources, organized as clusters located in specific sites. The target computing grid is a dedicated one, able to notify configuration changes such as jobs submission/ending. In our study, we considered a continuous stream of batch, independent jobs, i.e. jobs that don't need users interaction to be executed, and don't depend on the execution of other jobs. Moreover, we designed our scheduler in such a way it is able to manage the software licences, requested by the jobs to execute, as resources to be allocated. Eventually, we allowed users to specify a deadline for their jobs, this means that jobs can specify a time at which their execution has to be finished.

The Thesis work consists of two main research lines:

- the study of a high level scheduler, the Meta-Scheduler, exploited at the top of the hierarchy to dispatch jobs among the underlying clusters,
- the study of a low level scheduler, the Local-Scheduler, for which we investigated two solutions: one based on the Backfilling algorithm, and the other one based on the Convergent Scheduling technique.

The Thesis presents two important related results obtained by scientific collaborations. The former concerns to the application of the self-optimizing behavior to one job classification heuristics exploited at Meta-Scheduler level. The latter concerns to the comparison of our

Backfilling solution with a Schedule-Based solution proposed by Dalibor Klusáček and Hana Rudová of the University of Brno.

Concerning the study conducted for the scheduler at the top of the hierarchy, we proposed a Meta-Scheduler which aims to schedule arriving jobs balancing the workload among the underlying clusters, respecting the job running requirements and deadlines, and optimizing the utilization of hardware as well as software resources.

The proposed solution exploits a set of heuristics to classify incoming jobs, giving them a priority proportional to their relevance. Each heuristics manages a specific problem constraint (e.g. deadline and software licence requirements, submitting user peculiarities), and the computed job priority is used by the Meta-Scheduler in order to make scheduling decisions. Each heuristics works without any knowledge of the resources status, according to the on-line paradigm. The heuristics set is designed in order to be easy extensible. Future versions of the Meta-Scheduler classifier can be enriched with other heuristics in order to consider other QoS requirements.

The scheduling phase performed by our Meta-Scheduler is based on two functions: Load and Ordering. Load aims to balance the workload among clusters by assigning a job to the less loaded cluster. Ordering aims to balance the number of jobs with equal priority in each cluster queue.

One of our objectives was to design a lightweight Meta-Scheduler easy to interface with different scheduling algorithms exploited at Local-Scheduler level. This allows us to design future versions of our Meta-Scheduler able to support the cooperation with common grid schedulers such as: LSF (6), or PBS (7), or others. Moreover, as future work, we would like to extend the high level scheduling function set with a function which aim is to dispatch jobs to the underlying level in order to consider the size of each job. This because making scheduling decision for a set of small-size jobs is easier than for a set of big-size jobs. Our Meta-Scheduler is not able to prevent situation in which a set of big- or small-size jobs is assigned to a single Local-Scheduler. This may lead

to increase the number of jobs executing without respecting their QoS requirements. Otherwise, scheduling jobs also according to their size can improve the quality of the scheduling decisions.

At Local-Scheduler level we studied two different solutions: a Flexible Backfilling algorithm able to manage our specific constraints, and a new solution based on the Convergent Scheduling technique.

The proposed solutions were evaluated independently, by simulations, using different streams of jobs synthetically generated with different inter-arrival times between jobs. We haven't provided results with real data, because of the analyzed job streams strongly depends on the system configuration used to execute them, that is not representative for our system configurations. Therefore, real workload traces with deadlines and software requirements are not easy to find.

The proposed Flexible Backfilling (FB) strategy extends the EASY Backfilling algorithm, by exploiting a variety of heuristics to assign priorities to the queued jobs at each scheduling event (job submission/ending). FB is characterized by a classification phase, in which jobs are labeled with a priority value, and by a scheduling phase, in which jobs are assigned to the available machines. The job classification heuristics cover deadline requirements, license usage, aging (to prevent job starvation), and it favors the small-size jobs execution. We developed FB in order to classify incoming jobs exploiting dynamic information about both licences availability, and the jobs in the system. Concerning the scheduling phase, FB exploits dynamic information about jobs, licence availability, and machine status, to compute the scheduling plans.

We employed the FB algorithm at Local-Scheduler level, and we compare it with well know algorithms, such as FCFS, and an implementation of the EASY backfilling algorithm. Eventually, the Thesis presents results obtained comparing our FB with a Schedule-Based approach.

Besides the scientific results carried out using our Flexible Backfilling, it was a good opportunity to share research ideas, and to start a collaboration with other scientists. This collaboration improved my relational

capacities, and my scientific forming.

The Convergent Scheduling technique we proposed (CS) aims to schedule arriving jobs respecting their deadline requirements, and optimizing the exploitation of hardware and software resources.

The proposed scheduler exploits a set of heuristics, each one managing a specific problem constraint, which leads the scheduler in making decisions. To this end, the heuristics assign priorities to all the jobs in the system. The job priority measures the degree of preference of a job for each cluster's machine, i.e. how each machine suits well for the job execution. The scheduler aims to schedule a subset of queued jobs that maximize the degree of preference for the available resources. The jobs priorities are re-computed at each scheduling event. A new scheduling plan is carried out on the basis of the current status of the system, and information related both to jobs waiting for execution and running jobs. The modular structure of the proposed scheduler makes simple to define a set of customized heuristics in order to meet the administrator goals. The CS scheduler was evaluated comparing it with the Earliest Deadline First (EDF), and the Backfilling algorithms.

In future, we plan study the behavior of the Convergent Scheduling technique in real grid environments, and to enrich it with functionalities needed to address job migration, when different Local-Schedulers in the computing platform exploit our Convergent Scheduling algorithm.

The evaluation of the Multi-Level scheduling framework we proposed was carried out combining our Meta-Scheduler with the Flexible Backfilling we developed, and with our Convergent Scheduler.

To evaluate the interaction of Meta-Scheduler with the Flexible Backfilling we employed an ad hoc simulator we implemented. While, to evaluate the interaction of Meta-Scheduler with the CS, we employed the GridSim simulator. In both cases we synthetically generate the streams of jobs we used, this because of the most real streams of jobs we found in literature do not allow the job request of licences, and the job deadline time specification. As future work, we plan to study mechanisms, to

estimate the job execution time, exploiting a set of historical data describing the jobs features, and their running time. Experiments showed that our scheduler is able to dispatch jobs balancing the workload among clusters. Moreover, exploiting the Convergent Scheduling technique, our multi-level scheduling framework is able to execute a greater number of job respecting their QoS requirements, than using common algorithms such as EASY Backfilling, FCFS, or our Flexible Backfilling. Eventually, we plan to enrich our framework with mechanisms that enable the system to continue operating properly, even in the event of failure.

The Thesis also proposed a new design pattern to implement self-optimizing classifiers. We presented the components, and their interaction, that programmers have to implement when designing a self-optimizing classifier. Furthermore, we exploited our design pattern to implement a classifier, based on the Meta-Scheduler Deadline heuristics, as a case study. We showed the way our classifier adapts itself to different situations, and we compare its resulting classification with a not adapting classifier.

As future works, we plan to re-design our Meta-Scheduler classifier according to the self-optimizing design pattern we proposed. To meet this goal, we need to define a mechanism for the composition of self-optimizing heuristics, and this also implies the extension of the design pattern we proposed.

References

[1] The distributed asci supercomputer (das). http://www.cs.vu.nl/das2 Web site. 33

[2] The globus toolkit. http://www.globus.org/ Web site. 34, 35, 42

[3] Portable batch system. http://www.pbsgridworks.com Web site. 31

[4] Cluster resources^{inc}. http://www.clusterresources.com, 2001. 14, 36

[5] Viola vertically integrated optical testbed for large application in dfn. http://www.viola-testbed.de/ Web site, November 2005. 8, 105

[6] Platform lsf reports user's guide. http://www.platform.com Web site, October, 2005. 2, 128

[7] C.Lesiak B.Mann T.Proett A.Bayucan, R.L.Henderson and D.Tweten. Portable batch system: External reference specification. Technical report, RJ Technology Solutions, November 1999. 128

[8] David Abramson, Rajkumar Buyya, and Jonathan Giddy. A computational economy for grid computing and its implementation in the nimrod-g resource broker. *Future Gener. Comput. Syst.*, 2002. 14, 39

[9] R. Baraglia D. Puppin M. Pasquali L. Ricci A.D. Techiouba, G. Capannini. Backfilling strategies for scheduling streams of jobs on computational farms. In *In CoreGRID Workshop on Grid Programming Model, Grid and P2P Systems Architecture, Grid Systems, Tools and Environments*, June 12-13, 2007. 49, 63

[10] Altair. Pbs pro administrator guide. Web Site, 2004. 44

[11] Sudeepth Anand, Srikanth B. Yoginath, Gregor von Laszewski, Beulah Alunkal, and Xian-He Sun. Flow-based multistage co-allocation service. In *Communications in Computing*, 2003. 33

[12] M. Pasquali P. Dazzi A. Panciatici R. Baraglia. Self-optimizing classifiers: Formalization and design pattern. In *In Proceedings of the CoreGRID Symposium in conjunction with EuroPar 2008*, Las Palmas de Gran Canaria, Canary Island, Spain, August 2008. 104

[13] Yair Bartal, Amos Fiat, Howard Karloff, and Rakesh Vohra. New algorithms for an ancient scheduling problem. *J. Comput. Syst. Sci.*, 51(3), 1995. 14

[14] Fran Berman, Geoffrey C. Fox, and Anthony J. G. Hey. *Grid Computing Making the Global Infrastructure a Reality*. Wiley Series in Communications Networking & Distributed Systems, 2003. 12

[15] Francine Berman, Richard Wolski, Henri Casanova, Walfredo Cirne, Holly Dail, Marcio Faerman, Silvia Figueira, Jim Hayes, Graziano Obertelli, Jennifer Schopf, Gary Shao, Shava Smallen, Neil Spring, Alan Su, and Dmitrii Zagorodnov. Adaptive computing on the grid using apples. *IEEE Trans. Parallel Distrib. Syst.*, 14(4):369–382, 2003. 41

[16] Wojciech Bozejko and Mieczyslaw Wodecki. Solving the flow shop problem by parallel tabu search. In *PARELEC '02: Proceedings of the International Conference on Parallel Computing in Electrical Engineering*, Washington, DC, USA, 2002. IEEE Computer Society. 21

[17] Tracy D. Braun, Howard Jay Siegel, Noah Beck, Ladislau L. Bölöni, Muthucumaru Maheswaran, Albert I. Reuther, James P. Robertson, Mitchell D. Theys, Bin Yao, Debra Hensgen, and Richard F. Freund. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 2001. 23

[18] R. Buyya, D. Abramson, and J. Giddy. Economy driven resource management architecture for computational power grids. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA2000)*, Las Vegas,USA, 2000. 2, 47

[19] R. Buyya, J. Giddy, and D. Abramson. An evaluation of economy-based resource trading and scheduling on computational power grids for parameter sweep applications. 39

[20] R. Buyya and M. Murshed. Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. The Journal of Concurrency and Computation: Practice and Experience (CCPE), Wiley Press, May, 2002. 92, 98, 122

[21] Rajkumar Buyya, David Abramson, and Jonathan Giddy. Nimrod/g: An architecture for a resource management and scheduling system in a global computational grid, 2000. 39

[22] Luis M. Camarinha-Matos and Hamideh Afsarmanesh, editors. *Infrastructures for Virtual Enterprises: Networking Industrial Enterprises, IFIP TC5 WG5.3 / PRODNET Working Conference on Infrastructures for Virtual Enterprises (PROVE '99), October 27-28, 1999, Porto, Portugal*, volume 153 of *IFIP Conference Proceedings*. Kluwer, 1999. 13

[23] Nicolas Capit, Georges Da Costa, Yiannis Georgiou, Guillaume Huard, Cyrille Martin, Grgory Mouni, Pierre Neyron, and Olivier Richard. A batch scheduler with high level components. In *Cluster computing and Grid 2005 (CCGrid05)*, 2005. 3, 8, 30

[24] T. L. Casavant and J. G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Trans. Softw. Eng.*, 14(2):141–154, 1988. 13, 14

[25] Italian Interuniversity Consortium Cineca. http://www.cineca.it/, 2008. 6, 78, 92

[26] Karl Czajkowski, Ian Foster, and Carl Kesselman. Resource co-allocation in computational grids. In *HPDC '99: Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, page 37, Washington, DC, USA, 1999. IEEE Computer Society. 33

[27] H. Rudová R. Baraglia D. Klusáček, L. Matyska and G. Capannini. Local search for grid scheduling. In *Doctoral Consortium at the International Conference on Automated Planning and Scheduling (ICAPS'07)*, USA, 2007. 119

[28] L. Matyska D. Klusáček, H. Rudová. Alea - grid scheduling simulation environment. In *In the Proceedings of the 7th International Conference on Parallel Processing and Applied Mathematics (PPAM 2007), workshop on Scheduling for Parallel Computing*, LNCS. Springer-Verlag, 2008. 122

[29] Holly Dail, Henri Casanova, and Fran Berman. A decoupled scheduling approach for the grads program development environment. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–14, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press. 42

[30] Patrizio Dazzi, Francesco Nidito, and Marco Pasquali. New perspectives in autonomic design patterns for stream-classification-systems. In *WRASQ '07: Proceedings of the 2007 workshop on Automating service quality*, pages 34–37, New York, NY, USA, 2007. ACM. 9, 32, 103

[31] Olivier Delannoy, Nahid Emad, and Serge G. Petiton. Workflow global computing with yml. In *7th IEEE/ACM International Conference on Grid Computing*, pages 25–32, Barcelona, 2006. 3, 8, 27

[32] Fangpeng Dong and Selim G. Akl. Scheduling algorithms for grid computing: State of the art and open problems. Technical Report Technical Report No. 2006-504, School of Computing, Queens University, Kingston, Ontario, January 2006. 23

[33] Allen B. Downey and Dror G. Feitelson. The elusive goal of workload characterization. *SIGMETRICS Perform. Eval. Rev.*, 26(4):14–29, 1999. 6

[34] Carsten Ernemann, Volker Hamscher, Uwe Schwiegelshohn, Ramin Yahyapour, and Achim Streit. On advantages of grid computing for parallel job scheduling. In *CCGRID '02: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, Washington, DC, USA, 2002. IEEE Computer Society. 24, 26

[35] D. Feitelson, L. Rudolph, and U. Schwiegelshohn. Parallel job scheduling – a status report. In 10th Workshop on Job Scheduling Strategies for Parallel Processing, New-York, NY, June 2004. 10, 14, 16, 17, 18, 63

[36] Dror G. Feitelson. Parallel workloads archive. http://www.cs.huji.ac.il/labs/parallel/workload/, 2005. 6, 78, 92

[37] Larry; Schwiegelshohn Uwe Feitelson, Dror G.; Rudolph, editor. *Job Scheduling Strategies for Parallel Processing*, volume 2537 of *Lecture Notes in Computer Science*. Springer-Verlag, 8th International Workshop, JSSPP 2002, Edinburgh, Scotland, UK, July 24, 2002. 13

[38] Anja Feldmann, Jiří Sgall, and Shang-Hua Teng. Dynamic scheduling on parallel machines. *Theor. Comput. Sci.*, 130(1), 1994. 14

[39] Amos Fiat and Gerhard J. Woeginger, editors. *Online Algorithms, The State of the Art (the book grow out of a Dagstuhl Seminar, June 1996)*, volume 1442 of *Lecture Notes in Computer Science*. Springer, 1998. 13

[40] I. Foster. The anatomy of the grid: Enabling scalable virtual organizations. In *CCGRID '01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, page 6, Washington, DC, USA, 2001. IEEE Computer Society. 2

[41] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997. 34, 35, 42

[42] James Frey, Todd Tannenbaum, Miron Livny, Ian Foster, and Steven Tuecke. Condor-g: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3):237–246, 2002. 2

[43] D. Puppin L. Ricci M. Pasquali G. Capannini, R. Baraglia. A job scheduleing framework for large computing farms. In *SC 07 ACM/IEEE Computer Society International Conference for High Performance Computing, Networking, Storage, and Analysis*, 2007. 7, 10, 18, 49, 91, 93, 98

[44] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1995. 108

[45] M. R. Garey and Ronald L. Graham. Bounds for multiprocessor scheduling with resource constraints. *SIAM J. Comput.*, 4(2):187–200, 1975. 14

[46] Richard Gibbons. A historical application profiler for use by parallel schedulers. In *IPPS '97: Proceedings of the Job Scheduling Strategies for Parallel Processing*, pages 58–77, London, UK, 1997. Springer-Verlag. 7

[47] Fred Glover and M. Laguna. Tabu search. In C. Reeves, editor, *Modern Heuristic Techniques for Combinatorial Problems*, Oxford, England, 1993. Blackwell Scientific Publishing. 121

[48] Fred W. Glover and Gary A. Kochenberger. *Handbook of Metaheuristics (International Series in Operations Research & Management Science)*. Springer-Verlag, Berlin, January 2003. 118

[49] GENIAS Software GmbH. Codine: Computing in distributed networked environments, 1995. 2

[50] Volker Hamscher, Uwe Schwiegelshohn, Achim Streit, and Ramin Yahyapour. Evaluation of job-scheduling strategies for grid computing. In *GRID '00: Proceedings of the First IEEE/ACM International Workshop on Grid Computing*, London, UK, 2000. Springer-Verlag. 12

[51] Robert L. Henderson. Job scheduling under the portable batch system. In *IPPS '95: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 279–294, London, UK, 1995. Springer-Verlag. 44

[52] Kwok Y.K. Ahmad I. Dhodhi M. hmad, I. Scheduling parallel programs using genetic algorithms. In Ercal F. Olariu S. In Zomaya, A. Y., editor, *Solutions to Parallel and Distributed Computing Problems*, pages 231–254, New York, USA, 2001. 23

[53] Atsushi Hori, Hiroshi Tezuka, Yutaka Ishikawa, Noriyuki Soda, Hiroki Konaka, and Munenori Maeda. Implementation of gang-scheduling on workstation cluster. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 126–139. Springer-Verlag, 1996. 18

[54] E. S. H. Hou, N. Ansari, and H. Ren. A genetic algorithm for multiprocessor scheduling. *IEEE Trans. Parallel Distrib. Syst.*, 5(2):113–120, 1994. 23

[55] IBM. Autonomic Computing Initiative. www.ibm.com/autonomic. 104

[56] IBM. "autonomic computing: Ibm perspective on the state of information technology". http://www.research.ibm.com/autonomic/manifesto, 2001. 104

[57] J. P. Jones and C. Brickell. Second evaluation of job queuing/scheduling software: Phase 1 report. NAS Technical Report, NAS High Performance Processing Group, NASA Ames Research Center, Moffett Field, CA, June 1997. 2

[58] Waiman Chan Liana L. Fong Morris A. Jette Andy Yoo Jos E. Moreira, Hubertus Franke. *High-Performance Computing and Networking*, chapter A gang-scheduling system for ASCI blue-pacific, pages 829 – 840. Lecture Notes in Computer Science. Springer-Verlag, 1999. 18

[59] Ian Foster Carl Kesselman Karl Czajkowski, Steven Fitzgerald. Grid information services for distributed resource sharing. In *HPDC '01: Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing*, page 181, Washington, DC, USA, 2001. IEEE Computer Society. 42

[60] Keller, M. Hovestadt, O. Kao, and A. Streit. Scheduling in hpc resource management systems: Queuing vs. planning. In L. Rudolph D. G. Feitelson and U. Schwiegelshohn, editors, *In 9th International Workshop, Job scheduling strategies for parallel processing JSSPP*, Seattle, USA, June 24 2003. Springer-Verlag. 2, 118, 119

[61] Carl Kesselman and Ian Foster. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, November 1998. 12

[62] D. Klusáček, H. Rudová, R. Baraglia, M. Pasquali, and G. Capannini. Comparison of multi criteria scheduling techniques. In *In CoreGRID Integration Workshop 2008. Integrated Research in Grid Computing.*, Heraklion-Crete, Greece, April 2-4, 2008. 103, 118

[63] Jochen Krallmann, Uwe Schwiegelshohn, and Ramin Yahyapour. On the design and evaluation of job scheduling algorithms. In *IPPS/SPDP '99/JSSPP '99: Proceedings of the Job Scheduling Strategies for Parallel Processing*, pages 17–42, London, UK, 1999. Springer-Verlag. 14

[64] S. Krishnaswamy, S.W. Loke, and A. Zaslavsky. Estimating computation times of data-intensive applications. *Distributed Systems Online, IEEE*, 5(4):–, April 2004. 7

[65] Yi-Hsuan Lee and Cheng Chen. A modified genetic algorithm for task scheduling in multiprocessor systems. Taiwan: National Chiao Tung University, 2003. 23

[66] Joseph Leung, Laurie Kelly, and James H. Anderson. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press, Inc., Boca Raton, FL, USA, 2004. 20, 22

[67] Keqin Li and Yi Pan. Probabilistic analysis of scheduling precedence constrained parallel tasks on multicomputers with contiguous processor allocation. *IEEE Trans. Comput.*, 49(10):1021–1030, 2000. 14

[68] Li Wang Dawei Li. A scheduling algorithm for flexible flow shop problem. Dept. of Mathematics and Physics, Anshan Institute of Iron and Steel Technology Anshan, Liaoning, P. R. China. 21

[69] David A. Lifka. The anl/ibm sp scheduling system. In *IPPS '95: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, London, UK, 1995. Springer-Verlag. 14, 16

[70] Michael Litzkow, Miron Livny, and Matthew Mutka. Condor - a hunter of idle workstations. In *In Proceedings of the 8th International Conference of Distributed Computing Systems*, San Jose, CA, USA, 13-17 June 1988. 2

[71] Silvano Martello and Paolo Toth. *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc., New York, NY, USA, 1990. 67, 73

[72] Sun Microsystems. On demand computing: Using network.com. http://www.network.com, August 2007. White Paper. 1

[73] H. H. Mohamed and D. H. J. Epema. An evaluation of the close-to-files processor and data co-allocation policy in multiclusters. In *CLUSTER '04: Proceedings of the 2004 IEEE International Conference on Cluster Computing*, pages 287–298, Washington, DC, USA, 2004. IEEE Computer Society. 33

[74] H. H. Mohamed and D. H. J. Epema. Experiences with the koala co-allocating scheduler in multiclusters. In *CCGRID '05: Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 2*, pages 784–791, Washington, DC, USA, 2005. IEEE Computer Society. 3, 8, 33

[75] Sébastien Noël, Olivier Delannoy, Nahid Emad, Pierre Manneback, and Serge G. Petiton. A multi-level scheduler for the grid computing yml framework. In *Euro-Par Workshops*, pages 87–100, 2006. 3, 8, 27

[76] John K. Ousterhout. Scheduling techniques for concurrent systems. In *In proceedings of the Third International Conference on Distribuited Computing Systems*, 1982. 18

[77] Andrew J. Page and Thomas J. Naughton. Dynamic task scheduling using genetic algorithms for heterogeneous distributed computing. In *IPDPS*

'05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 6, page 189.1, Washington, DC, USA, 2005. IEEE Computer Society. 23

[78] P. Dazzi A. Panciatici M. Pasquali. A formal checklist for self-optimizing strategy-driven priority classification system. Technical Report 2008-TR-005, ISTI-CNR, February 2008. 104

[79] P. Dazzi M. Pasquali. Formalization of autonomic heuristics-driven systems. Technical Report 2008-TR-004, ISTI-CNR, February 2008. 104

[80] Ludek Matyska Pavel Fibich and Hana Rudová. Model of grid scheduling problem. In AAAI Press Technical Reports, editor, *In AAAI05 Workshop on Exploring Planning and Scheduling for Web Services, Grid and Autonomic Computing*, 2005. 2, 3

[81] Dimitrios Pendarakis, Jeremy Silber, and Laura Wynter. Autonomic management of stream processing applications via adaptive bandwidth control. In *ICDCS '06: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, Washington, DC, USA, 2006. IEEE Computer Society. 9

[82] Michael L. Pinedo. *Planning and Scheduling in Manufacturing and Services (Springer Series in Operations Research and Financial Engineering)*. Springer-Verlag, Berlin, June 2005. 118

[83] Diego Puppin, Mark Stephenson, Walter Lee, and Saman Amarasinghe. Convergent scheduling. *The Journal of Instruction Level Parallelism*, 6(1):1–23, September 2004. 49

[84] Cluster Resources. Portable batch system, administrator guide. Web Site, 1998. 44

[85] M. Pasquali R. Baraglia G. Capannini D. Laforenza L. Ricci. A two-level scheduler to dynamically schedule a stream of batch jobs in large-scale grids. HPDC ACM/IEEE International Symposium on High Performance Distributed Computing, Boston, US, June 23-27 2008. Poster. 3, 5, 18

[86] Uwe Schwiegelshohn and Ramin Yahyapour. Analysis of first-come-first-serve parallel job scheduling. In *SODA '98: Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, Philadelphia, PA, USA, 1998. Society for Industrial and Applied Mathematics. 16, 63

[87] J. Sgall. On-line scheduling – a survey. In A. Fiat and G. Woeginger, editors, *On-Line Algorithms*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1997. 4, 21, 47

[88] Warren Smith, Ian T. Foster, and Valerie E. Taylor. Predicting application run times using historical information. In *IPPS/SPDP '98: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 122–142, London, UK, 1998. Springer-Verlag. 7

[89] I. Foster L. Liming J. Link W. Allcock, J. Bresnahan and P. Plaszczac. Gridftp update. Technical report, January 2002. 34

[90] Yair Wiseman and Dror G. Feitelson. Paired gang scheduling. *IEEE Trans. Parallel Distrib. Syst.*, 14(6):581–592, 2003. 19

[91] Oliver Wldrich, Wolfgang Ziegler, and Philipp Wieder. A meta-scheduling service for co-allocating arbitrary types of resources. Technical Report TR-0010, Institute on Resource Management and Scheduling, CoreGRID - Network of Excellence, December 2005. 8, 105

[92] Edward Xia, Igor Jurisica, Julie Waterhouse, and Valerie Sloan. Scheduling functional regression tests for ibm db2 products. In *CASCON '05: Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research*, pages 292–304. IBM Press, 2005. 7

[93] Dan Tsafrir Yoav Etsion. A short survey of commercial cluster batch schedulers. School of Computer Science and Engineering, Jerusalem, The Hebrew University, Israel. 38

[94] Albert Y. Zomaya and Yee-Hwei Teh. Observations on using genetic algorithms for dynamic load-balancing. *IEEE Trans. Parallel Distrib. Syst.*, 12(9):899–911, 2001. 23

[95] Albert Y. Zomaya, Chris Ward, and Ben Macey. Genetic scheduling for parallel processor systems: Comparative studies and performance issues. *IEEE Trans. Parallel Distrib. Syst.*, 10(8):795–812, 1999. 23