A Multi-criteria Job Scheduling Framework for Large Computing Farms

Ranieri Baraglia[a,*], Gabriele Capannini[a], Patrizio Dazzi[a], Giancarlo Pagano[b]

[a] *Information Science and Technology Institute - CNR Pisa (Italy)*
[b] *Department of Computer Science - University of Pisa (Italy)*

Abstract

In this paper, we propose a new multi-criteria job scheduler for scheduling a continuous stream of batch jobs on large-scale computing farms. Our solution, called Convergent Scheduler, exploits a set of heuristics that drives the scheduler in taking decisions. Each heuristics manages a specific problem constraint, and contributes to compute a value that measures the degree of matching between a job and a machine. Scheduling choices are taken both to meet the Quality of Service requested by the submitted jobs and optimize the usage of software and hardware resources. In order to validate the scheduler we propose, it has been compared versus two common job scheduling algorithms: Easy and Flexible backfilling. Convergent Scheduler demonstrated to be able to compute good assignments that allow a better exploitation of resources with respect to the other algorithms. Moreover, it has a simple a modular structure that makes simple its extension and customization to meet the service goal of an installation.

Keywords: multi-objective optimization, job scheduling, heterogeneous computing farm, utility computing, performance evaluation.

1. Introduction

Nowadays, most on-demand computing resources provisioning systems, both in the form of Cloud and Utility computing, are structured as computing farms. Computing farms consists of heterogeneous hardware and software resources, such as workstations, parallel machines, storage arrays, and software licenses, networked together by a middleware to provide transparent, remote, and secure access to computing power. This middleware allows machines to coordinate their activities and to share the hardware, software, and data system's resources.

In a computing farm, users submit their computational requests without necessarily knowing on which computational resources these will be executed. Moreover, requests may require different types and levels of Quality of Service

*Corresponding author
Email address: r.baraglia@isti.cnr.it (Ranieri Baraglia)

(QoS). A typical QoS requirement is the time at which user wants to receive results, i.e. the job turnaround time.

At middleware level the scheduler is fundamental to achieve high performance. The goal of a scheduler is to assign computational resources to address all the application and installation requirements and/or constraints. It has to answer questions like: what machine can be assigned to a job? How long? Does this assignment allow maximizing the resource utilization? Does it allow to satisfy the 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 [31]: 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). The general problem of finding an optimal scheduling has been demonstrated to be NP-complete[31]. Moreover, in the computational environment we consider, dynamic and fast scheduling decisions are necessary to meet jobs' QoS requirements[16].

In the past, a lot of research effort has been devoted to develop job scheduling architectures and algorithms [3, 15, 14]. Different classification can be made when schedulers are analyzed [13, 6]: 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. on-line and offline), according to the resulting schedule (e.g. queue-based and schedule-based). 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. In the distributed model, schedulers interact among themselves in order to decide which resources should be assigned to the jobs to be executed. In such model there is not a central leader responsible for scheduling, hence it is potentially highly scalable and fault-tolerant. However, because the status of remote jobs and resources are not available in a single location; the generation of optimal schedules 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 a combination of the centralized and distributed models. The scheduler at the top of the hierarchy is called meta-scheduler; it interacts with the local schedulers to make scheduling decisions. This model well suits for grid environments.

According to static scheduling algorithms 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 re-evaluation 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. Online scheduling algorithms process the incoming jobs without to have knowledge on the whole input job stream. They take decisions for each arriving job not knowing future input, while offline algorithms know all the jobs

before taking scheduling decisions. Queue-based scheduling algorithms take as input jobs stored in a queue, and tries to assign them to available resources. Schedule-based approach allows precise mapping of jobs onto machines in time. Due to their computational cost, these approaches were mostly applied to static problems, assuming that all the jobs and resources are known in advance, which allows to create schedule for all jobs at once [2, 12].

Many of the proposed algorithms are currently exploited both in commercial and open source job schedulers, which are mostly based on the queuing systems of various types that are designed with respect to specific needs. However, at our knowledge, none of the proposed schedulers is able to deal with a wide range of constraints and requirements (e.g. number of licenses concurrently usable and job deadlines) presented by both users and providers.

In this paper we propose a centralized on-line multi-criteria job scheduling framework, called Convergent Scheduler (CS), to dynamically assign a continuous stream of batch jobs on the machines of large computing farms made up of heterogeneous, single-processor or SMP nodes, linked by a low-latency, high-bandwidth network.

A first version of CS was proposed in [4], a previous our work. CS exploits a set of heuristics that guide the scheduler in making decisions. Each heuristics manages a specific problem constraint, and contributes to compute a value that measures the degree of matching between a job and a machine. The scheduler aims to schedule arriving jobs respecting their deadline, and optimizing the exploitation of hardware and software resources. CS permits us to compute an effective job-scheduling plan, on the basis of the current resource availability and the information related to executing jobs as well as jobs waiting for execution. In order to take decisions, the scheduler assigns priorities to all the jobs in the system; then the computed priority value are used to determine a matching.

In order to improve both the number of jobs executed respecting their constraints and the resource usage the first CS version was extended by introducing a greedy algorithm, which incrementally computes the job-machine associations according to the space sharing policy, and considering the constraints on the license usage. With respect to the CS first version, it permits us to remove the assumption of running only a job per machine. The adoption of the greedy algorithm reduces the scheduler complexity saving the possibility to build effective scheduling plans.

To manage the problem constraints, we revised and extended the set of heuristics proposed in the CS first version. In order to make the proposed heuristics more effective, they are normalized in such a way that the value computed by each heuristics is in the range $[0, 1]$. It makes job priorities more trustworthy with respect to the first scheduler version.

The modular structure of CS makes simple to define a set of customized heuristics in order to meet the service goal of an installation. Heuristics refinements and extensions to support, for example, energy efficiency could enhance CS. Dispatching workloads to more energy-efficient machines could do it; less energy-efficient machine could be released saving energy.

The proposed scheduler was evaluated by simulations using the GridSim sim-

ulator [1], which permits us to simulate a computing farm varying the number of jobs, and machines and licenses availability.

The rest of this paper is organized as follows. Section 2 describes some of the most common job scheduling algorithms and frameworks. Section 3 gives a description of the problem. Section 4 describes our solution, and Section 5 describes the Flexible Backfilling algorithm we used in the experimental tests. Section 6 outlines and evaluates the proposed job scheduler through simulation. Finally, conclusions are given in Section 7.

2. Related work

In the past, a major effort has been devoted to understand and develop jobs scheduling algorithms and frameworks on distributed systems that are classified according to several taxonomies [13, 6]. The algorithms for batch jobs scheduling can be divided into two main classes: on-line and offline. Examples of on-line scheduling algorithms are: First-Come-First-Serve [14, 28], Backfilling [23, 11], List Scheduling [25]. Examples of offline algorithms are: SMART [27] and Preemptive Smith Ratio Scheduling [26].

Many of these algorithms are exploited into commercial and open source job schedulers [10], such as Maui[1] scheduler, IBM LoadLever[2], Load Sharing Facility[3], Portable Batch System[4], Oracle Grid Engine[5], previously known as Sun Grid Engine.

In the following, we give a description of the Backfilling algorithm because it is exploited in the most common commercial and open source job schedulers, such as Maui scheduler, Portable Batch System and Sun Grid Engine, and we also use it in our solution. Moreover, some recognized relevant job scheduling frameworks exploiting in cluster computing are described.

Backfilling is an optimization of the First-Come First-Served (FCFS) algorithm. Backfilling tries to balance the goals of resource utilization, maintaining the FCFS jobs order. As matter of fact, FCFS guarantees that jobs are started in the order of their arrivals. 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 each job to specify 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 at 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

[1] http://www.clusterresources.com/products/maui-cluster-scheduler.php
[2] http://www.redbooks.ibm.com/abstracts/sg246038.html
[3] http://www.platform.com/workload-management/high-performance-computing
[4] http://en.wikipedia.org/wiki/Portable_Batch_System
[5] http://www.oracle.com/technetwork/oem/grid-engine-166852.html

will never completely violate the FCFS order where some jobs are never run (a phenomenon known as starvation). In order to improve performance, some backfilling variants have been proposed by introducing some parameters, such as number of reservations, order of the queued jobs, and lookahead into the queue: EASY (Extensible Argonne Scheduling sYstem) [24], Conservative, and Flexible backfilling [11].

In the original EASY backfilling algorithm, jobs may be scheduled out of order only if they do not delay the job at the top of the queue. For such job, the scheduler estimates when a sufficient number of resources will be available and reserves them for it. Backfilling-based strategy proposed in [33] exploits adaptive reservations as a function of the job delay, due to previous backfilling decisions.

The Flexible variant was obtained from the original EASY backfilling by prioritizing jobs according to a scheduler goals. Jobs are then queued according to their priority value, and selected for scheduling (including candidates for backfilling) according to this priority order [33, 5].

The Conservative variant tries to reduce the delays in the execution of the waiting jobs by making reservations for all jobs. However, simulation results indicate that delaying jobs down in the queue is rarely a problem, and that Conservative backfilling tends to achieve reduced performance in comparison with EASY. Other backfilling variants exploit adaptive reservations or consider the amount of lookahead into the queue [29].

The OAR[6] job scheduler has been designed as an open platform for research and experiments. The main contribution of OAR is its design, based on two high-level components: a SQL database, 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 by using SQL queries, and to store and exploit logging and accounting information. 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. Although making SQL queries might induce some overhead compared to a specific implementation, the OAR engine has demonstrated a good behavior under high workload. In OAR each queue has its own scheduling policy, and the order among jobs, stored in the same queue, is based on their arrival time.

The IBM LoadLever supports various scheduling algorithms such as FCFS, FCFS with backfilling, gang scheduling [35], and can also be customized with specific external scheduling algorithms provided by the installation. The system supports the job checkpoint/restart functionality. It also allows customizing features such as the computation of job priorities, the grouping of jobs into classes according to their characteristics and/or processing requirements, and the calculation of fairness. By default, the selection priority of a job is a function of its submission time to the system, according to FCFS technique. In addition, LoadLever allows the job preemption to interrupt the execution of low priority

[6] `http://oar.imag.fr`

job to support the highest priority ones. LoadLever supports backfilling with the ability to specify the metric (e.g. BestFit or Firstfit) used to choose the job for backfilling. When the backfilling technique is used, jobs exceeding their estimated execution time are interrupted.

Load Sharing Facility (LSF) is a resource manager for clusters released by Platform Computing Corporation. Its primary objective is to maximize resource utilization within the constraints imposed by the management policies adopted by individual installations. LSF allows the management of both interactive and batch jobs, and supports several scheduling algorithms, such as FCFS and Backfilling, and can interface with scheduling algorithms defined by the installation. LSF organizes the submitted jobs in priority queues, with the possibility to define a different scheduling algorithm for each queue. This allows managing different QoS in different queues. To meet the QoS of jobs, LSF also adopts advanced resource reservation techniques, and a Service Level Agreement functionality that allows an installation to define the scheduling objective (e.g. job deadlines, system throughput). The FCFS algorithm is the default behavior of LSF. When the Backfilling algorithm is adopted for a queue, the preemption is not applied to backfilled jobs to prevent that such jobs delay the completion of their execution at the expense of the job in front of the queue for which a resource reservation was made. The LSF's fair-share algorithm allows the installation to divide users into groups and allocate resources to users according to a priority value defined by the installation. In LSF job starvation is treated by dynamically managing job priorities. According to the policy of an installation, the priority of a job can increase during his stay in the system. Moreover, LSF provides functionality for balancing the workload among the compute nodes. Based on the system workload, jobs can be dynamically migrated between processing nodes or rescheduled. In addition to "traditional" resources such as CPU, memory and storage space, LSF also manages software licenses by automatically forwarding jobs to licensed hosts, or by holding jobs in batch queues until licenses are available.

Maui is a centralized open source job scheduler for batch jobs designed to maximize the resource usage and minimize the average job turnaround. It supports the concept of job class (or job queue). Within Maui, all jobs are associated with a class, and each class may have an associated set of constraints determining what types of jobs can be submitted to it. Furthermore, Maui computes the job priorities as a weighted sum of several factors, and each factor itself is a weighted sum of sub-factors. Both weighted factors and weighted sub-factors are governed by the installation. Maui can exploit different queue scheduling algorithms. However, its default behavior is a simple FCFS batch scheduler with a backfilling policy. Moreover, to make scheduling decisions Maui uses a fair-share technique based on historical data describing previous job executions. In Maui, to meet jobs QoS, an extensive control over which jobs are considered eligible for scheduling, how the jobs are prioritized, when a job preemption take place, and where jobs are performed is allows to site administrators.

The Portable Batch System (PBS) is a batch queuing and job management system. It comes in two flavors: OpenPBS, which is intended for small clusters,

and PBS-Pro, which is the industrial strength version [22]. PBS-Pro provides mechanisms for submitting batch job requests on or across multiple machines, and submitted jobs are assigned to the properly queue according to their attributes. Jobs eligible for execution are scheduled on a server node according to the jobs queue priority and available resources and requirements. PBS-Pro default scheduler policies are FCFS, Shortest Job First (SJF), user/group priorities, and fair-share. Moreover, it is extensible with the integration of own site schedulers. It allows to define different queue scheduling policies. The fair-share scheduler uses a hierarchical approach similar to the LSF one. Other features include checkpoint/restart, preemption, backfilling and advanced resources reservation. Job preemption is provided 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. Since the use of SJF could lead to job starvation, to avoid this situation, PBS-Pro adopts the backfilling algorithm, which is also used to meet the jobs deadline.

Oracle Grid Engine (OGE) is an open source queued job scheduler supported by Oracle. OGE goals are to increase machines and application licenses productivity at the same time as optimizes the number of jobs that can be executed. Submitted jobs are prioritized according to their computational requirements (e.g. memory size, processor execution speed, requested software licenses), and then are assigned to the appropriate queue. The queues are characterized by the technical properties of the server nodes on which are located. It allows to execute a job on the node able to provide the corresponding service. Jobs in a queue are selected according to their priority values. OGE supports the FCFS (the default one) and a fair-share scheduling policy. The latter is used to distribute resources equally among the users and it is an optional administrator set function. SGE supports check-pointing and server node-to-server node job migration.

Despite their wide application, none of the proposed schedulers can deal with the entire range of constraints, such as number of floating licenses concurrently usable and job deadlines. In general, to meet the various QoS levels, the available job schedulers apply a different scheduling criterion to each job queue. This way, the various scheduling criteria are managed separately. Even if the multi-criteria approach seems to be a proper technique to efficiently solve the scheduling problem on heterogeneous and distributed computing platforms, only a few research solutions has been proposed. In [9, 7, 8] a bicriteria algorithm for scheduling jobs on clusters is shown. It exploits two pre-selected criteria for minimizing job makespan as well as their average completion time. Siddiqui *et al.* [30] propose a negotiation-based scheduling mechanisms exploiting advanced reservation to satisfy users' requirements, which are defined as utility functions on the base of the values and negotiation levels specified by the users. In [20] the authors presented user preference driven multi-objective scheduling strategies for Grids. The idea presented in [20] has then been extended to multi-stakeholder case and further developed in [19, 21, 18]. In the framework we propose, there is only a submission queue, and a job-machine

association is carried out by exploiting the contributions of a set of heuristics which one managing a problem constraint. Moreover, the modularity of the proposed scheduling algorithm allows managing a large set of requirements and constraints coming from both user jobs and service providers by increasing the number of heuristics exploited to compute the degree of matching between a job and a machine.

3. Problem description

In our study we consider a continuous stream of batch jobs (a set of jobs). Jobs and machine are enumerated respectively by the sets N and M. We suppose that a job $i \in N$ is sequential or multi-threaded, that it is executed only on a single machine $m \in M$, 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. Since we assume that jobs are preemptable, we model three types of job preemption: stop/restart, suspend/resume, checkpoint/restart [32]. Moreover, we consider that all the machines support stop/restart, a part of them can also support suspend/resume, while checkpoint/restart is exploitable only if the job is properly instrumented, and explicitly required by a user. A job execution may be interrupted whenever a schedule event takes place (i.e. job submission and ending).

Furthermore, let L be the set of available licenses. Each job can require a subset $L' \subseteq L$ of licenses to be executed. A license is called floating if it can be used on different machines, non-floating if it is bound to a specific machine. At any time, the number of used licenses must not be greater than their availability. In our study, we consider the association of licenses to machines. As a consequence, when more jobs require the same license and are executed on the same machine, only one license copy is accounted.

Jobs are annotated with information describing their computational requirements and constraints. Machines and licenses are annotated with information describing their hardware/software features. A job is described by an identifier, its submission and deadline time, an estimation of its duration, a benchmark score, the number and type of processors and licenses requested. Since Gridsim does not model intra-site communications, the cost due to moving a job input data set to a machine or between machines is not considered in this work. A benchmark score and the number and type of CPUs describe a machine. The benchmark score is used for jobs as well as for machines. For a machine, it specifies the computational power expressed in MIPS. For a job, it specifies the computational power of the machine used for estimating its execution time. To estimate the number of instruction in a job, the MIPS value related to the reference machine is multiplied for the estimated job completion time. This time can be evaluated statistically by analyzing historical data or by benchmarking.

Furthermore, each software license is defined in term of its type as well as its availability. The type defines if a license can be enabled only on a fixed subset of machines or if it can be concurrently used on a variable subset of them. The

availability defines the maximum number of licenses that can be enabled at the same time in the farm.

The building of a scheduling plan involves all the $|N|$ jobs in the system (queued + running), and the scheduler aims to schedule arriving jobs respecting their deadline requirements, and optimizing license and machine usage.

4. Convergent scheduling for batch job scheduling

The current solution exploits a matrix $P^{|N|\times|M|}$, called Job-Machine matrix, where each entry (i, m) stores the priority value $p_{i,m}$, which specifies the degree of preference of the job i for the machine m. As shown in Figure 1 the current scheduling framework is structured according to two main phases: Heuristics and Matching.

In our case N changes dynamically as a new job is submitted to the system at any time. In such a context, in order to meet job deadlines and avoid wasting resources, fast scheduling decisions are needed to prevent computational resources from remaining idle, and scheduling decisions from aging.

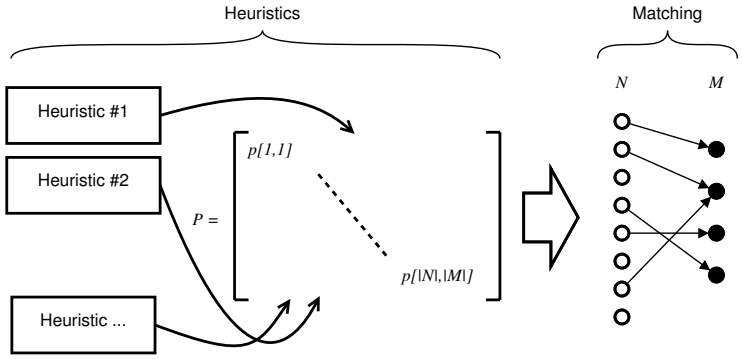

Figure 1: Structure of the Convergent Scheduling framework.

The Job-Machine matrix constitutes the common interface to heuristics. Each heuristics changes priority values in order to compute jobs-machines matching degrees. The final priority value $p_{i,m}$ of a job i with respect to an eligible machine is the weighted sum of the values $\Delta_{p_{i,m}}$ computed by each heuristics. Managing the constraints by using distinct heuristics, and structuring the framework in subsequent phases, leads to a modular structure that makes it simple to extend and/or to modify. The value returned by each heuristics is normalized in such a way that it is in the range $[0, 1]$. It makes job priorities more reliable with respect to the first scheduler version.

The Matching phase elaborates the resulting P matrix to carry out a new scheduling plan, i.e. a set of job-machine associations. Each matrix element expresses the preference degree in selecting a machine m to run a job i. The aim of the matching algorithm is to compute the associations job-machine to which correspond to a larger preference degree according to the problem constraints.

4.1. Collection of heuristics

Six heuristics have been exploited to build our CS framework. The order of magnitude of the heuristics complexity is O($|N| \cdot |M|$).

∘ *Minimal requirements.* This heuristics fixes the job-machines associations. It selects the set 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 the floating licenses. For each job $i \in N$ and for all the machines $m \in M_{Aff_i}$ the following heuristics are computed to build a scheduling plan.

To implement the different heuristics we estimate the remaining job execution time on a machine m on which the job will be executed as:

$$T_{remaining_{i,m}} = \frac{RemainInstr_{i,m}}{MIPS_m} \tag{1}$$

where $RemainInstr_{i,m}$ is the number of instructions that has still to be executed in order to complete the execution of the job i on the machine m, and $MIPS_m$, as the number of million of instruction per second a machine m is able to elaborate.

∘ *Deadline.* The Deadline heuristics aims to compute the Job-Machine update value $\Delta_{p_{i,m}}$ in order to execute jobs respecting their deadline. For each job i, it requires an estimation of its execution time, in order to evaluate its completion time with respect to the current wall clock. Jobs close to their deadline get a boost in priority that gives them a scheduling advantage. The priorities values $p_{i,m}$ are increased proportionally with respect to their closeness to the deadline when executed on a machine m. For each job i, first we compute:

$$T_{last_{i,m}} = Deadline_i - k \cdot T_{remaining_{i,m}} \tag{2}$$

$$T_{end_{i,m}} = Wallclock + T_{remaining_{i,m}} \tag{3}$$

where $T_{last_{i,m}}$ is the estimation of latest time at which the job i should begin its execution on the machine m to address its deadline $Deadline_i$, $Wallclock$ is the current simulator time, and $T_{end_{i,m}}$ indicates the estimated time at which i will end its execution on m. k is a constant value fixed by the installation. It permits us to overestimate a job execution time. With k = 1, i.e. without overestimation, any event able to delay the execution of a job would lead to violate its deadline (in our test k was fixed equal to 1).

Then, the function $f()$ is computed as:

$$f(i,m) = \begin{cases} 0 & \text{if } T_{end_{i,m}} \leq T_{last_{i,m}} \\ \Gamma & \text{if } T_{last_{i,m}} < T_{end_{i,m}} \leq Deadline_i \\ 1 & \text{if } T_{end_{i,m}} > Deadline_i \end{cases} \tag{4}$$

where Γ is given by:

$$\Gamma = \frac{T_{end_{i,m}} - T_{last_{i,m}}}{T_{remaining_{i,m}}} \tag{5}$$

The equation 4 defines the "urgency" of running a job i with respect to a machine m. High values of $f_{i,m}$ indicates that i executed on m has a high probability to be executed does not respecting its deadline. Then, the values to update the job-machine matrix entries are computed according to the following expressions:

$$\Delta_{D_{p_{i,m}}} = W_D \cdot (1 - f_{i,m}) \times F_i \tag{6}$$

where W_D indicates the weight of the heuristics, and F_i is computed as::

$$F_i = \frac{1}{|M_{Aff_i}|} \cdot \sum_{\forall m \in M_{Aff_i}} f_{i,m} \tag{7}$$

The value F_i indicates the average urgency value of job i, computed evaluating the urgency of job i on all the machines able to run it, whereas $(1 - f_{i,m})$ is used for ordering the results on the basis of the computational power of the considered machines. Figure 2 gives a graphical representation of the Deadline heuristics. The heuristics aims to assign a minimum value, i.e. 0, to jobs 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, $T_{last_{i,m}}$, the score assigned to the job is increased in inverse proportion with respect to such distance. The contribution of Deadline increases until $T_{end_{i,m}}$ reaches the job deadline, i.e. $\Delta_{p_{i,m}}$ is increased at a maximum score, i.e. 1. When $T_{end_{i,m}}$ is greater than the job deadline, the contribution of the heuristics is null, i.e. 0.

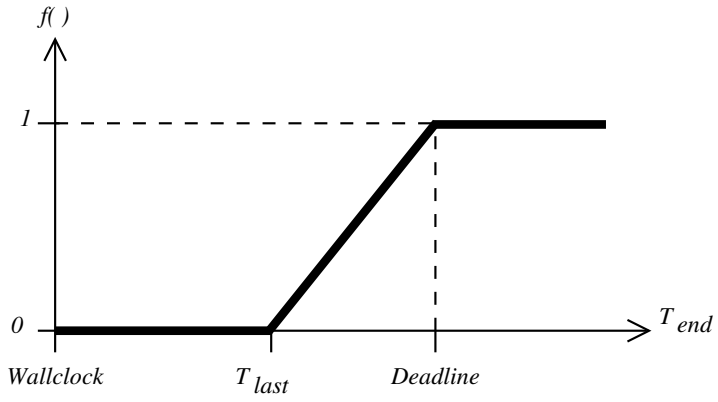

Figure 2: Graphical representation of the Deadline heuristics.

∘ *Overhead minimization.* Three kinds of preemption are modeled: stop/restart, suspend/resume and checkpoint/restart. A job preemption take place at a schedule event, that is job submission or ending, when a new scheduling plan is computed.

Since a job preemption may happen at any schedule event, this heuristics aims to minimize the waste in execution time due to the interruption of a job execution on a machine, and its resuming or restarting on another one, or on the same one at a different time. The heuristics tries to minimize job migrations by increasing the priorities of the job-machine associations present in the current scheduling plan. $\Delta_{p_{i,m}}$ is computed as:

$$\Delta_{p_{i,m}} = W_C \cdot \frac{T_{elapsed_{i,m}}}{max(T_{remaining_{i,m}}, T_{elapsed_{i,m}})} \tag{8}$$

where W_C is the heuristics weight, and $T_{elapsed_{i,m}}$ is the elapsed execution time of a job i on machine m.

∘ *Licenses.* This heuristics updates the job priorities to favor the execution of jobs that increase the critical degree of licenses. A license becomes critical when there is a number of requests greater than the available number of its copies. The Δ values to update the Job-Machine matrix entries are computed as a function of the number of critical and not critical licenses requested by a job. The greater the number of critical licenses is, the greater Δ is. This pushes jobs using many licenses to be scheduled first, this way releasing a number of licenses.

To identify the critical licenses, we introduce the ρ_l parameter defined as:

$$\rho_l = \frac{\sum_{i=1}^{|N|} s_{i,l}}{a_l} \tag{9}$$

with:

$$s_{i,l} \in \{0,1\} \ \wedge \ s_{i,l} = 1 \text{ iff } i \text{ asks for } l \in L \tag{10}$$

where the numerator specifies the "request", i.e. how many jobs are requesting a license, and the denominator a_l specifies the "offer", i.e. how many copies of a license can be simultaneously active on the system machines. A license is considered critical if $\rho_l > 1$.

Let $L_i^{'} = \{l \in L \mid i \text{ asks for } l\}$ the set of licenses required by a job i.

The value $\Delta_{p_{i,m}}$ is computed as:

$$\Delta_{p_{i,m}} = W_L \cdot min\left(\frac{\sum_{l=1}^{|L|} s_{i,l} \cdot \rho_l}{|L_i^{'}|}, 1\right) \tag{11}$$

where W_L is the weight of the heuristics.

∘ *Wait minimization.* The aim of this heuristics is to minimize the average time jobs spend waiting to complete their execution. It tries to minimize this time by advancing the jobs that have a shorter estimated time to complete their execution. $\Delta_{p_{i,m}}$ is computed as a function of the time remaining to end the execution of a job i on each machine $m \in M_{Aff_i}$.

$$\Delta_{p_{i,m}} = W_{WM} \cdot \left(1 - \frac{T_{remaining_{i,m}}}{T_{sup_m}}\right) \tag{12}$$

where W_{WM} is the weight of the heuristics, and T_{sup} is computed as:

$$T_{sup_m} = max\{T_{remaining_{i,m}}\}$$

∘ *Anti-aging.* The goal of this heuristics is to avoid that a job remains, for a long time, waiting to start or progress its execution. The value to update the Job-Machine entries is computed as a function of the "age" that a job has reached in the system. $\Delta_{p_{i,m}}$ is computed as:

$$\Delta_{p_{i,m}} = W_{AA} \cdot \frac{Age_i}{Age_i + T_{remaining_{i,m}}} \tag{13}$$

where W_{AA} is the weight of the heuristics, and Age_i is equal to the difference between the current simulation time and the time at which the job has been submitted.

In this work, we hand-tuned the weight parameters of the heuristics, but their best setting could be computed by using an evolutionary framework according to the installations objectives.

4.2. Matching

This phase is devoted to build a scheduling plan, and it is executed after the Job-Machine matrix has been updated by all the heuristics. It elaborates the resulting P matrix to carry out the job-machine associations, that is the new scheduling plan. Each matrix element expresses the preference degree in selecting a machine m to run a job i. The aim of the matching algorithm is to compute the job-machine associations to which correspond to a larger preference degree, according to the problem constraints. To this end a greedy algorithm is adopted. The elements of P are arranged in descending order, and the obtained array is visited starting from the first element to fix the association between a job and a machine. The new scheduling plan is built by incrementally defining the job-machine associations, and updating the number of processors available on the machines as well as the number of available licenses. An association job-machine takes place if the selected machine has enough free processors and can run a copy of the licenses needed to execute the selected job. The licenses are a limited resource; their availability could decrease setting up the scheduling plan. Even if this way does not guarantee a matching which corresponds to the maximum system usage, it leads to carry out an effective scheduling plan, i.e. the system will have enough computational resources to run the selected job subset.

To order P two different sorting algorithm were experimented: Quicksort and Counting sort [17]. The second one has a smaller computational complexity when the items to sort are discretized in classes. However, to discretize in classes the job priority values means to use coarser priority value than those used by Quicksort. Therefore, the use of the Counting sort leads to better execution time, but to less quality in computing a scheduling plan, with respect to of using Quicksort.

5. The Flexible Backfilling algorithm

In this section we present the Flexible Backfilling (FB) algorithm we used to evaluate the quality of the solutions carried out by CS. The structure of 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. We developed FB in such a way it is able to classify incoming jobs in order to handle the jobs QoS requirements. It uses dynamic information about both licenses and jobs to carry out the job classification. Furthermore, it exploits dynamic information about jobs, licenses, and machines to compute the scheduling plans. For instance, FB increases the job priorities according to the time jobs spent waiting for execution. 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 within its deadline), or a requested license 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 subset of previous described heuristics to assign a priority value p_i to each job i: Anti-Aging, Deadline, Wait Minimization and License. Licenses is the general heuristics already presented, Anti-aging, Deadline, and Minimization 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:

$$p_i = \Delta_{AA} + \Delta_D + \Delta_L + \Delta_{WM}$$

∘ *Anti-aging.* It computes Δ_i as fallows:

$$\Delta_i = W_{AA} \cdot Age_i$$

with Age_i computed as:

$$Age_i = Wallclock - T_{submit_i}$$

where T_{submit_i} is the job submission time.

∘ *Deadline.* This heuristics comes from that previously described. It computes the T_{end_i}, i.e. the time at which FB estimates the termination of the job i, as follows:

$$T_{end_i} = Wallclock - T_{estimate_i}$$

$$T_{last_i} = Deadline - (K \cdot T_{estimate_i}) \; with \; k \geq 0$$

$$\alpha_i = \frac{Max - Min}{T_{last_i}}$$

where $T_{estimate_i}$ is the estimated job completion time compute with respect to the more power machine $m \in M_{Aff_i}$. The Min and Max values are defined by system administrators. They are the weights of the heuristics and determine the angular coefficient α_i of the straight line passing through the points of coordinates (T_{last_i}, Min) and $(Deadline_i, Max)$. Δ_{p_i} is increased according to the function described by such line, and is computed as:

$$\Delta_{p_i} = \begin{cases} min & \text{if } T_{end_i} \leq T_i \\ \alpha_i \cdot (T_{end_i} - T_{last_i}) & \text{if } T_{last_i} < T_{end_i} \leq Deadline_i \\ min & \text{if } T_{end_i} > Deadline_i \end{cases} \quad (14)$$

∘ *Wait Minimization.* The aim of this heuristics is to reduce the time jobs waiting to be executed. It is made by pushing small jobs to be scheduled first, this way to release resources that could be reserved for other larger jobs. The heuristics computes Δ_{p_i} as.

$$\Delta_{p_i} = W_{WM} - \frac{T_{Min}}{T_{estimate_i}}$$

where T_{Min} is the estimated execution time of the smaller queued job.

The scheduling phase is carried out applying the backfilling strategy, which makes a resource reservation for the first queued job. Queued jobs are arranged in decreasing order with respect to their priority values. To schedule a job i, FB arranges the machines belonging to M_{Aff_i} 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 jobs response time.

6. Performance evaluation

In this section we present the results obtained in the experiments conducted to evaluate the feasibility of CS. Comparing scheduling algorithms is generally a difficult task because of the different underlying assumptions in the original

studies of each algorithm. Moreover, public available archives contain jobs described by using only few parameters, typically the job submission and execution times, and the number of processors used.

In our study, we focus on job deadline and license usage, which, at our knowledge, are not present in any available job archive. Therefore, our evaluation was conducted by simulations using different streams of jobs generated according to an exponential distribution with different inter-arrival times between jobs. Moreover, each job and machine parameter was randomly generated from a uniform distribution.

We evaluated four different versions of the proposed scheduler, called CS, CS-Cs, CS-Td and CS-CsTd. CS uses the Quicksort algorithm representing the priority values as float numbers. CS-Cs uses the Counting sort algorithm representing the priority values as integers in the range [1, 1024]. CS-Td and CS-CsTd are the time driven version of CS and CS-Cs, respectively. They compute a new scheduling plan every 10 seconds.

For our evaluation, the GridSim simulator has been used. In order to simulate the computational environment considered in our study, GridSim has been modified by adding new subtypes that extend the behavior of the GridSim's classes *Machine*, and *Gridlet*. The first one has been extended to specify the subset of usable licenses and the type of preemption exploitable on a machine as well as the possibility to verify the compatibility of a job with a machine. Such compatibility is satisfied if the machine has a number of processors greater than or equal to that required by the job, and if it supports the licenses request by the job. To the second one has been added the support for deadlines, preemption and licenses. The third one has been extended to support the tracing of the parameters associated with a job life cycle, such as submission, execution and waiting times.

We simulated a computing farm varying the number of jobs, machines and licenses availability. For each simulation, we randomly generated a list of jobs and machines whose parameters are generated according to a uniform distribution in the ranges: estimated $[500 \div 3000]$, deadline $[25 \div 150]$, number of CPUs $[1 \div 8]$, benchmark $[200 \div 600]$, license ratio $[55\% \div 65\%]$, license suitability 90%, license needs 20%.

To each license we associate the parameter "license ratio" that specifies its maximum number of copies concurrently usable. The benchmark parameter is chosen for both jobs and machines. "license suitability" specifies the probability that a license is usable on a machine. "license needs" specifies the probability that a job needs a license. 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 the jobs are elaborated.

The weight associated to the heuristics were hand-tuned to give more importance to the job deadline requirement, the license usage constraints, and to

contain the number of operations of job preemption[7].

To evaluate the CS scheduler, tests were conducted by simulating a cluster of 150 machines, 20 licenses, and 1500 jobs, with the 2% of jobs supporting checkpoint/restart. In order to obtain stable values, each simulation was repeated 20 times with different job attributes values.

To evaluate the quality of schedules computed by CS, we exploited the following criteria: the percentage of jobs that do not respect the deadline, the percentage of machine and license usage and the job slowdown. Moreover, to evaluate the efficiency of CS, we evaluated the scheduling execution time and the scheduler scalability.

The evaluation was conducted by comparing our solution with an EASY Backfilling (Easy BF), and a Flexible Backfilling (Flexible BF) algorithms. For Flexible BF the job priority values are computed at the computation of a new scheduling plan. Computing the priority value at each new scheduling event permits us to meet the scheduler goals [34] in a better way.

Each job is scheduled to the best available machine, respecting the constraints on both the number of CPUs and of licenses. If an eligible machine is not found the job is queued and re-scheduled at the next step of the scheduling process. It exploits the jobs preemptive feature to free resources to execute first new submitted job with an earliest deadline.

To evaluate the schedules carried out by the CS schedulers we generated six streams of jobs with jobs inter-arrival times (Ta in Figure 3) fixed equal to 1, 4, 6, 8, 10, and 12 simulator time unit ($1 \div 12$ in Figures 4, 5, 6, 7, 8, 9).

As shown in Figure 3 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 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.

During the simulation the execution time of a job i on a machine m is computed as:

$$runtime_{i,m} = \frac{totalInstr_i}{MIPS_m} \tag{15}$$

where $totalInstr_i$ is the number of instructions of i expressed in millions and $MIPS_m$ is the benchmark value associated to m.

Figure 4 shows the percentage of jobs executed do not respect their deadline. 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

[7] Heuristics weight values used in our tests: Deadline ($W_D = 15.0$), Overhead Minimization ($W_C = 40.0$), Anti-aging ($W_{AA} = 5.0$), Licenses ($W_L = 5.0$), Wait minimization ($W_{WM} = 8.0$).

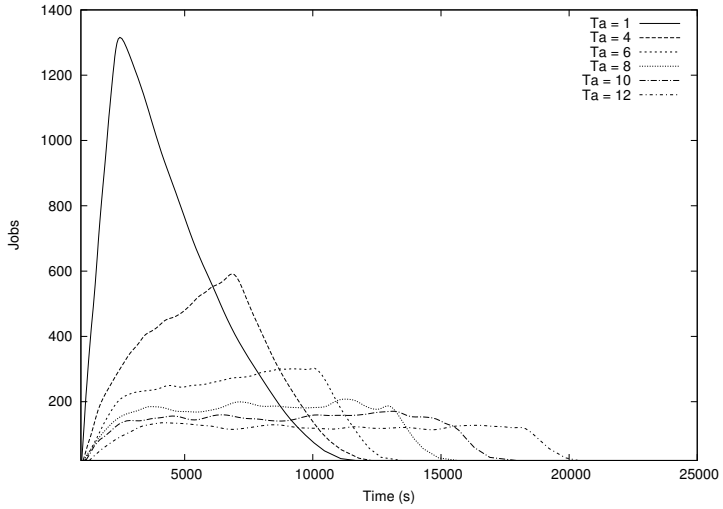

Figure 3: System workload through simulations.

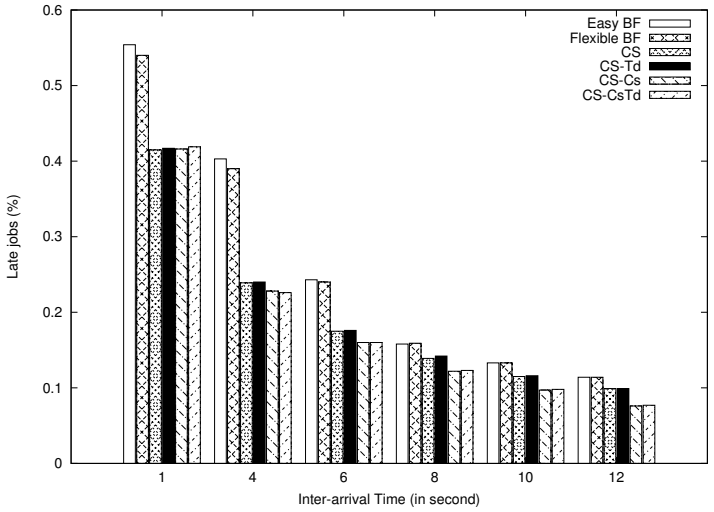

Figure 4: Percentage of jobs executed do not respecting their deadline.

increases. Satisfactory results are obtained when the available cluster computational power is able to maintain almost constant the system contention. The backfilling-based algorithms obtain worse results than those obtained by CS. In particular, CS obtains significative better results when the job competition in the system increases, i.e. for T_a equal to 1 and 4.

Figure 5 shows the job slowdown, which is computed as $(Tw+Te)/Te$, with Tw the time that a job spends waiting to start and/or restart its execution,

and Te the job execution time [24]. This measure figures out as the system load delays the execution of such jobs. As can be seen, for job streams generated with an inter-arrival time greater than $T_a = 1$, jobs scheduled by using CS obtained a Tw equal or very close to 1.

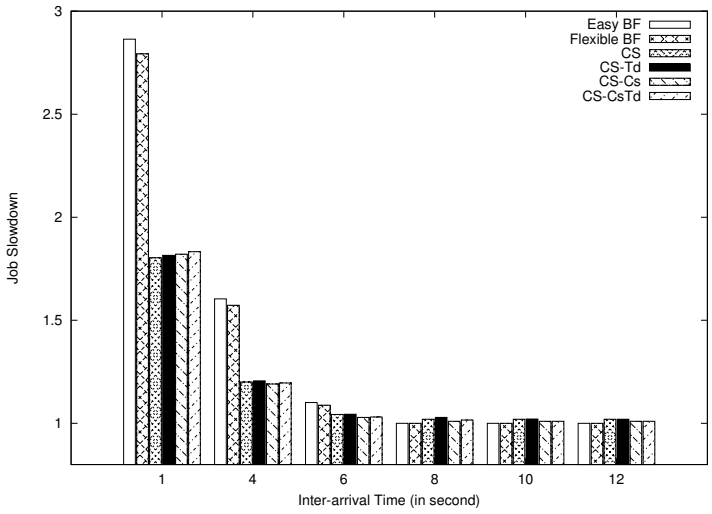

Figure 5: Job slowdown.

In Figure 6, the percentage of machine usage is shown. It is computed as:

$$\overline{SU} = \frac{\sum_{i=0}^{n-1}(SU_{t_i} \times (t_{i+1} - t_i))}{t_n} \tag{16}$$

where n is the simulation length, and SU_i is given by:

$$SU_i = \frac{\sharp Processors_{active}}{min(\sharp Processors_{available}, \sharp Processors_{required})} \tag{17}$$

which is computed at each time t_i when the building of a new scheduling plan is completed.

A machine can remain idle when it does not support all the hardware/software requirements of a queued job. We can see that CS schedules jobs in a way that permits to maintain all the system machines always busy.

In Figure 7, the percentage of license usage is shown.

It is computed likewise $\overline{SU}$, more in detail:

$$LU_{t_i} = \frac{\sharp Licenses_{active}}{\min(\sharp Licenses_{total}, \sharp Licenses_{requested})} \tag{18}$$

The average usage is computed at the end of the simulation, according to this equation:

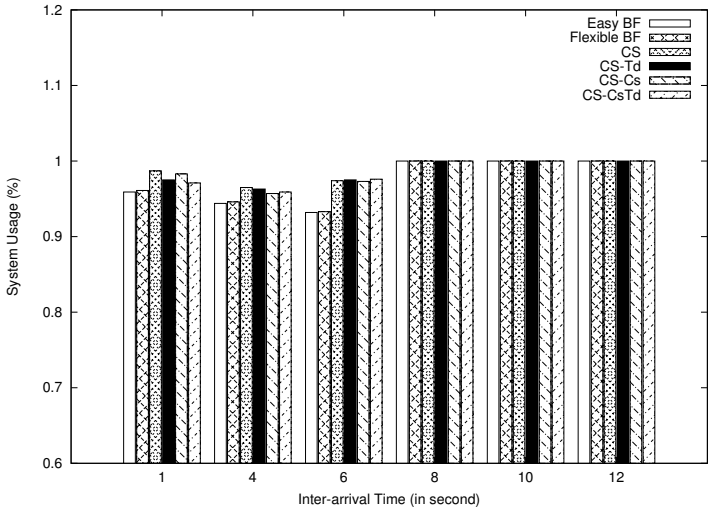

Figure 6: Percentage of machine usage.

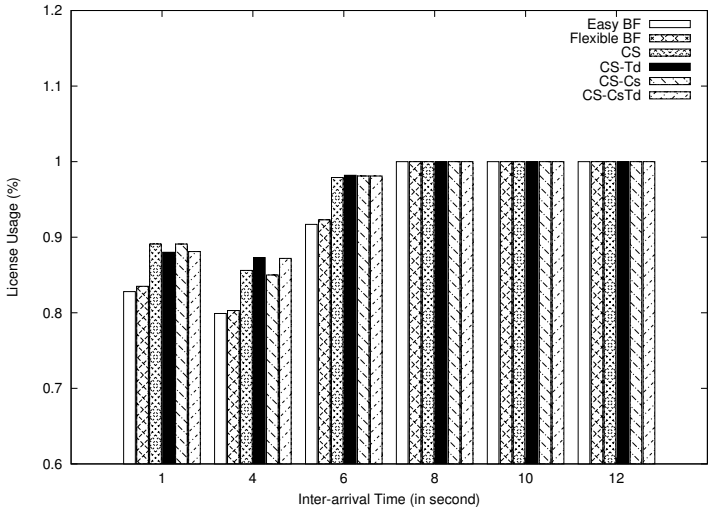

Figure 7: Percentage of license usage.

$$\overline{LU} = \frac{\sum_{i=0}^{n-1}(LU_{t_i} \cdot (t_{i+1} - t_i))}{t_n} \tag{19}$$

It can be seen that when the jobs contention decreases all the algorithms are able to efficiently use the available floating licenses. However, CS obtains a better license usage for T_a equal to 1, 4 and 6.

Figures 8 and 9 show respectively the average time spent to build a new scheduling plan and the total scheduling times spent to schedule each job stream. As expected, the CS versions, which supports the preemption feature, require more execution time than the backfilling-based algorithms. The best times are obtained by the CS-CsTd version.

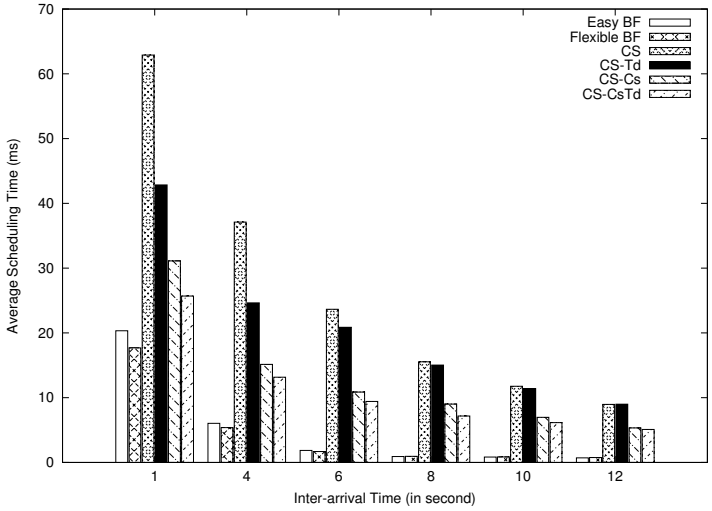

Figure 8: Average scheduling times.

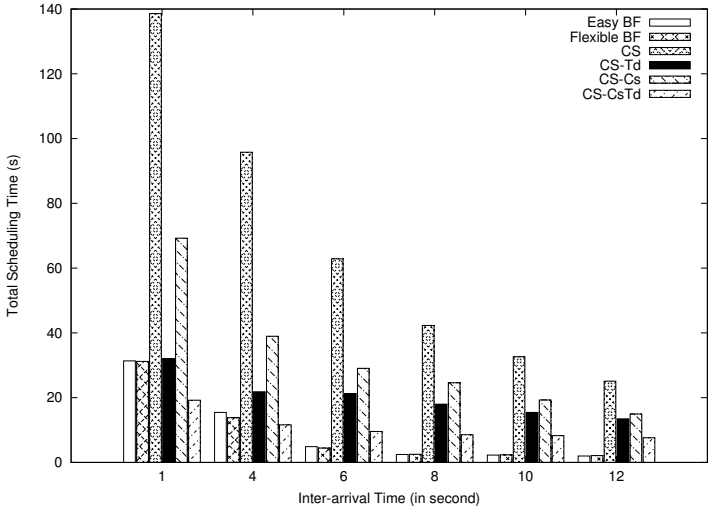

Figure 9: Total scheduling times.

In Figure 10 the scalability for the CS and Flexible Backfilling algorithms

is shown. The schedulers scalability was evaluated measuring the time needed to carry out a new scheduling plan increasing the number of jobs. It can be seen that CS achieves a good scalability. In particular the better scalability was obtained by the CS-Cs, which has a smaller time complexity than the CS version.

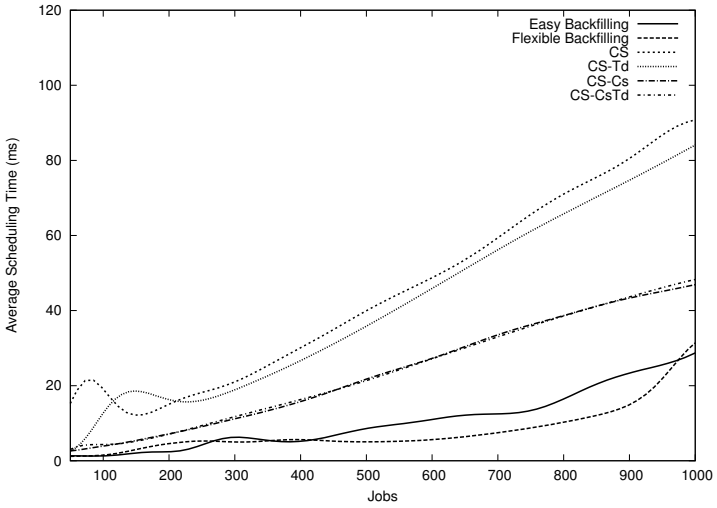

Figure 10: Algorithm scalability.

By exploiting the jobs preemption, to favor the execution of higher priority jobs, and properly setting the heuristics contribution to the job priorities, CS is able to build scheduling plans with a good trade-off between the addressing of job QoS requirements and the minimization of the costs induced by the preemption of jobs.

7. Conclusion

In this paper we propose Convergent Scheduler (CS), a multi-criteria job scheduler able to dynamically schedule a continuous stream of batch jobs on the machines of large computing farms, made up of heterogeneous, single-processor or SMP nodes, linked by a low-latency, high-bandwidth network. It aims at scheduling jobs respecting their deadline, and optimizing the machine and license usage. The proposed scheduler exploits a set of heuristics, each one managing a specific problem constraint, that guide the scheduler in making decisions. To this end the heuristics assign priorities to all the jobs in the system, and jobs priorities are computed to build a scheduling plan, which is computed on the basis of the current status of the system and information related to jobs waiting for execution. Four different versions of CS, implemented by using the Quicksort and the Counting sort algorithms, were evaluated by simulations using the Gridsim simulator. The evaluation was conducted by using different streams of jobs

generated according to an exponential distribution with different inter arrival times between jobs. Moreover, each job and machine parameter was randomly generated from a uniform distribution. Due to the different manners to represent the priority values, CS versions exploiting the Quicksort algorithm obtains the best scheduling quality, whereas the best scheduling execution times are obtained by the CS versions exploiting the Counting sort algorithm. Moreover, CS was evaluated comparing it with two different Backfilling schedulers: EASY and Flexible. The conducted evaluation analyzing the behavior and quality of the presented approach demonstrated that the proposed solution is a viable one. Moreover, the modular structure of CS makes simple to define a set of customized heuristics in order to meet the service goal of an installation. CS could be enhanced by including heuristics refinements and extensions to support, for example, energy efficiency. It could be done by dispatching workloads to more energy-efficient machines. So less energy-efficient machine could be released saving energy.

Acknowledgment

This work has been supported by SUN Microsystems's grant, and by the European CoreGRID NoE (European Research Network on Foundations, Software Infrastructures and Applications for Large Scale, Distributed, GRID and Peer-to-Peer Technologies, contract no. IST-2002-004265).

References

[1] Anthony, S., Uros, C., Srikumar, V., Borut, R., Rajkumar, B., September 2008. A toolkit for modeling and simulating data grids: An extension to gridsim. Concurrency and Computation: Practice and Experience 20 (13), 1591–1609.

[2] Armentano, V. A., Yamashita, D. S., 2000. Tabu search for scheduling on identical parallel machines to minimize mean tardiness. Journal of Intelligent Manufacturing 11, 453–460.

[3] Buisson, J., Sonmez, O., Mohamed, H., Lammers, W., Epema, D., September 2007. Scheduling malleable applications in multicluster systems. In: Proceedings of IEEE Cluster '07. pp. 372–381.

[4] Capannini, G., Baraglia, R., Puppin, D., Ricci, L., Pasquali, M., 2007. A job scheduling framework for large computing farms. In: SC. p. 54.

[5] Chiang, S.-H., Arpaci-Dusseau, A. C., Vernon, M. K., Lect. Notes Comput. Sci. vol. 2537, 2002. The impact of more accurate requested runtimes on production job scheduling performance. In: JSSPP '02: Revised Papers from the 8th International Workshop on Job Scheduling Strategies for Parallel Processing. Springer-Verlag, London, UK, pp. 103–127.

[6] Dong, F., Akl, S. G., January 2006. Scheduling algorithms for grid computing: State of the art and open problems. Tech. Rep. 2006-504, School of Computing, Queen's University Kingston, Ontario.

[7] Dutot, P. F., Eyraud, L., Mounié, G., Trystram, D., 2004. Bi-criteria algorithm for scheduling jobs on cluster platforms. In: SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures. ACM, New York, NY, USA, pp. 125–132.

[8] Dutot, P.-F., Eyraud, L., Mounié, G., Trystram, D., 2005. Scheduling on Large Scale Distributed Platforms: From Models to Implementations. International Journal of Foundations of Computer Science 16 (2), 217–237.

[9] Dutot, P. F., Trystram, D., May 2005. A best-compromise bicriteria scheduling algorithm for malleable tasks. In: Press, C. (Ed.), Workshop on Efficient Algorithms. Santorini Island, Greece.

[10] Etsion, Y., Tsafrir, D., May 2005. A short survey of commercial cluster batch schedulers. Tech. Rep. 2005-13, School of Computer Science and Engineering, The Hebrew University of Jerusalem.

[11] Feitelson, D. D., Rudolph, L., Schwiegelshohn, U., LNCS vol. 3277, 2004. Parallel job scheduling, a status report. In: Job Scheduling Strategies for Parallel Processing 10th International Workshop. Springer-Verlag, London, UK, pp. 1–16.

[12] Feldmann, A., Sgall, J., Teng, S.-H., 1994. Dynamic scheduling on parallel machines. Theor. Comput. Sci. 130 (1), 49–72.

[13] Fiat, A., Sep. 1998. Online Algorithms: The State of the Art (Lecture Notes in Computer Science). Springer.

[14] Hamscher, V., Schwiegelshohn, U., Streit, A., Yahyapour, R., Lect. Notes Comput. Sci. vol. 1971, 2000. Evaluation of job-scheduling strategies for grid computing. In: GRID '00: Proceedings of the First IEEE/ACM International Workshop on Grid Computing. Springer-Verlag, London, UK, pp. 191–202.

[15] Karatza, H. D., January 2006. Scheduling gangs in a distributed system. International Journal of Simulation: Systems, Science & Technology 7 (1), 15–22.

[16] Klusek, D., Rudov, H., Baraglia, R., Pasquali, M., Capannini, G., 2008. Comparison of multi-criteria scheduling techniques, In: Grid Computing Achievements and Prospects. Springer.

[17] Knuth, D. E., 1998. The art of computer programming, volume 3: (2nd ed.) sorting and searching. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

[18] Kurowski, K., Nabrzyski, J., Oleksiak, A., Weglarz, J., 2004. Multicriteria aspects of grid resource management. J. of Scheduling, 271–293.

[19] Kurowski, K., Nabrzyski, J., Oleksiak, A., Weglarz, J., October 2008. A multicriteria approach to two-level hierarchy scheduling in grids. J. of Scheduling 11, 371–379.

[20] Kurowski, K., Nabrzyski, J., Pukacki, J., 2001. User preference driven multiobjective resource management in grid environments. In: CCGRID '01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid. IEEE Computer Society, Washington, DC, USA, p. 114.

[21] Kurowski, K., Oleksiak, A., Weglarz, J., October 2010. Multicriteria, multiuser scheduling in grids with advance reservation. J. of Scheduling 13, 493–508.

[22] Lambert M. Surhone, Miriam T. Timpledon, S. F. M., July 2010. Portable Batch System. Betascript.

[23] Lifka, D. A., Lect. Notes Comput. Sci. vol. 949, 1995. The anl/ibm sp scheduling system. In: IPPS '95: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing. Springer-Verlag, London, UK, pp. 295–303.

[24] Muálem, A. W., Feitelson, D. G., June 2001. Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. IEEE Trans. Parallel and Distributed Syst. 12 (6), 529–543.

[25] Radulescu, A., van Gemund, A. J. C., 2002. Low-cost task scheduling for distributed-memory machines. IEEE Trans. Parallel Distrib. Syst. 13 (6), 648–658.

[26] Schwiegelshohn, U., 2004. Preemptive weighted completion time scheduling of parallel jobs. SIAM Journal on Computing 33 (6), 1280–1308.

[27] Schwiegelshohn, U., Ludwig, W., Wolf, J. L., Turek, J., Yu, P. S., 1999. Smart bounds for weighted response time scheduling. SIAM J. Comput. 28 (1), 237–253.

[28] Schwiegelshohn, U., Yahyapour, R., 1998. Analysis of first-come-first-serve parallel job scheduling. In: SODA '98: Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, pp. 629–638.

[29] Shmueli, E., Feitelson, D. G., 2005. Backfilling with lookahead to optimize the packing of parallel jobs. J. Parallel Distrib. Comput. 65 (9), 1090–1107.

[30] Siddiqui, M., Villazón, A., Fahringer, T., 2006. Grid capacity planning with negotiation-based advance reservation for optimized qos. In: SC'06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing. ACM, New York, NY, USA, p. 103.

[31] Sinnen, O., 2007. Task Scheduling for Parallel Systems. Wiley-Interscience, Englewood Cliffs, New Jersey.

[32] Snell, Q., Clement, M. J., Jackson, D. B., 2002. Preemption based backfill. In: JSSPP '02: Revised Papers from the 8th International Workshop on Job Scheduling Strategies for Parallel Processing. Springer-Verlag, London, UK, pp. 24–37.

[33] Srinivasan, S., Kettimuthu, R., Subramani, V., Sadayappan, P., Lect. Notes Comput. Sci. vol. 2537, 2002. Selective reservation strategies for backfill job scheduling. In: JSSPP '02: Revised Papers from the 8th International Workshop on Job Scheduling Strategies for Parallel Processing. Springer-Verlag, London, UK, pp. 55–71.

[34] Techiouba, A., Capannini, G., Baraglia, R., Puppin, D., Pasquali, M., Ricci, L., 2008. Backfilling strategies for scheduling streams of jobs on computational farms. In: Making Grids Work. Springer US, pp. 103–115. URL http://dx.doi.org/10.1007/978-0-387-78448-9_8

[35] Wiseman, Y., Feitelson, D. G., 2003. Paired gang scheduling. IEEE Trans. Parallel Distrib. Syst. 14 (6), 581–592.

Ranieri Baraglia graduated in Computer Sciences in 1982 at the University of Pisa, Italy. He is currently a senior researcher at the Information Science and Technologies Institute of Italian National Research Council. Ranieri Baraglia was a contract professor at the Department of Mathematics at Perugia University from 1991 to 1996, and at the Department of Computer Science at Pisa University in 1999. His current research interests include high performance computing, job scheduling, P2P computing, Web mining. More information at http://hpc.isti.cnr.it/~ranieri/.

Gabriele Capannini was born in Italy in 1977. Currently he is PhD student of the Computer Science Department of the University of Pisa. Gabriele Capannini is currently affiliated to the High Performance Computing Laboratory (HPC Laboratory) of the Information Science and Technologies Institute (ISTI) of the Italian Research Council (CNR). He has published many international and national research papers and attended different international conferences. His current research interests include scheduling problem, efficiency in Information Retrieval, diversification techniques for Web Search Engine query results, and computational models. Visit http://hpc.isti.cnr.it/~gabriele/ for more information.

Patrizio Dazzi is a researcher of the High Performance Computing Laboratory at the Information Science and Technologies Institute, part of the Italian National Research Council (ISTI-CNR). He graduated (BSc equivalent) in Computer Science in 2003, then he obtained in 2004 the Laurea Specialistica degree (MSc equivalent) in Computer Technologies and the PhD in Computer Science and Engineering in 2008. His research interest is primarly in Parallel and Distributed Systems. In particular, he is focusing its activities in Peer-to-Peer systems and Cloud computing. He is currently involved in the Contrail EU Project.

Giancarlo Pagano obtained his Bachelor's degree in Computer Science from University of Pisa in July 2009. He worked as a trainee in the High Performance Computing Laboratory of the National Research Council in Pisa, focusing his work on scheduling algorithm in large computing farm. He is pursuing his M.Sc. in Computer Science from University of Pisa.