Noname manuscript No.
(will be inserted by the editor)

A Multilevel Scheduler for Batch Jobs on Large-scale Grids

Marco Pasquali · Ranieri Baraglia · Gabriele Capannini · Laura Ricci · Domenico Laforenza

Received: date / Accepted: date

Abstract This paper proposes a two-level scheduler for dynamically scheduling a continuous stream of sequential and multi-threaded batch jobs on large-scale grids, made up of interconnected clusters of heterogeneous single-processor and/or symmetric multiprocessor machines. The scheduler aims to schedule arriving jobs respecting their computational and deadline requirements, and optimizing the hardware and software resource usage. At the top of the hierarchy a lightweight Meta-Scheduler classifies incoming jobs according to their requirements, and schedules them among the underlying resources balancing the workload. At cluster level a Flexible Backfilling algorithm carries out the job machine associations by exploiting dynamic information about the environment. Scheduling decisions at both the levels are based on job priorities computed by using different sets of heuristics. The different proposals have been compared through simulations. Performance figures show the applicability of our approach.

Keywords Multicriteria Job scheduling · Meta-scheduler · Workload Balancing · Grid

1 Introduction

To build a grid infrastructure requires the development and deployment of middleware, services, and tools. At middleware level the scheduler plays a major role in order to efficiently and effectively schedule submitted jobs on the available resources. The

Marco Pasquali
Information Science and Technologies Institute, Italian National Research Council, Pisa, Italy
E-mail: marco.pasquali@gmail.com

Ranieri Baraglia · Gabriele Capannini
Information Science and Technologies Institute, Italian National Research Council, Pisa, Italy
E-mail: {marco.pasquali, ranieri.baraglia, gabriele.capannini}@isti.cnr.it

Ricci Laura
Department of Computer Science, University of Pisa, Pisa, Italy E-mail: ricci@di.unipi.it

Domenico Laforenza
Institute of Informatics and Telematics, Italian National Research Council, Pisa, Italy E-mail: domenico.laforenza@iit.cnr.it

objective of the scheduler is to assign tasks to specific resources maximizing the overall resource utilization and guaranteeing the QoS required by the applications. In general, we can distinguish three different scheduling architectures: Centralized, Distributed, and Hierarchical [11], [2]. The centralized ones can be used for managing single or multiple resources located either in a single or multiple domains. It can only support uniform scheduling policy, and suits well for cluster management. This kind of scheduler architecture is not suitable for grids. The distributed one exploits a distinct scheduler to manage the machines belonging to a site. In this model, the schedulers interact among themselves to decide on which resources to allocate jobs being executed.

However, since the information describing the status of the remote resources and jobs is not available at a single scheduler, to build efficient schedules can be a not easy task. This scheduler architecture is suitable for grids. The hierarchical model is also suitable for grids. It 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 is considered to be the more suitable for grid environments.

In this paper we describe the study conducted to develop a two-level scheduler to dynamically schedule a continuous stream of independent batch jobs in large-scale grids. We investigated a novel solution to schedule jobs on a set of clusters by balancing the workload and respecting the job QoS requirements. The scheduler aims to schedule arriving jobs respecting their deadlines, and optimizing the utilization of hardware resources as well as software resources.

At the top of the hierarchy, a *Meta-Scheduler* assigns a priority value to each job, and then dispatches jobs to the underling clusters by means of two scheduling functions.

At cluster level, a Flexible Backfilling algorithm [6] is adopted as local schedulers.

The rest of the paper is organized as follows. Section 2 describes some of the most common job schedule solutions. Section 3 gives a description of the scheduling problem we consider. Section 4 describes the solution we propose. Section 5 outlines and evaluates the proposed solutions through simulations. Finally, conclusion and future work are described in Section 6.

2 Related work

In the past, a lot of research effort was devoted to understand and develop job scheduling algorithms [7], [8], [10], [12] that are classified according to several taxonomies [5], [9].

EASY Backfilling [14] is one of the common used scheduling algorithm. It requires that each job specifies an estimation of its execution time. It selects queued jobs according to the First-Come-First-Served (FCFS) policy [16], and in case of resource shortage to execute the first queued job, it makes for such job a resource reservation. Jobs back in the queue may be scheduled out of order only if they do not delay the job at the top of the queue. A variant of Backfilling is the *Flexible Backfilling* one. This variant was obtained from the original EASY Backfilling by prioritizing jobs according to scheduler goals. Jobs are then queued according to their priority values, and selected for scheduling (including candidates for backfilling) according to this priority order.

In the literature different schedule approaches exploiting a hierarchical architecture are proposed. In [15], a multi-level scheduler integrated in the YML framework is described. YML focuses on two major aspects: the development of parallel applications and their execution on grid environments. It has multiple objectives:

– to schedule a set of YML components with input data and precedence constraints issued from one or more users;
– to provide computing resources for these components in a multi-middleware environment;
– to offer users a guarantee in terms of the application completion time;
– to dynamically reorganize the schedule if unexpected events occur.

YML is based on an economic approach on resources, and defines different entities (e.g. resource provider/consumer) that interact within the grid infrastructure.

OAR [4] is a multilevel batch scheduler based on Backfilling. Jobs priorities are managed through submission queues.

Each queue has its own admission rules, scheduling policy and priority. Reservations are a special case in which users ask for a time slot on a specific resource. The jobs schedule is computed by a module called "meta-schedule" which manages reservations and queues.

KOALA [13] is a two-level grid scheduler that assigns to each submitted job a priority value that could be high or low. The job priority is used to determine the importance of a job relative to other jobs in the system w.r.t. their QoS requirements. The submitted jobs can belong to the low-priority queue or the high-priority queue. KOALA scans the queues to find jobs that can be scheduled.

VIOLA [1], [17] is characterized by a meta-scheduling service, which is able to co-allocate tasks on different resources in multiple domains. The meta-scheduler can interact with different scheduling systems exploited in each domain. Its main function is to negotiate the reservation of accessible resources, which are managed by their respective local scheduling systems. The goal of the negotiation is to determine a common time slot where all required resources are available to start a job execution.

3 Problem description

In our study, we consider a continuous stream of batch jobs, which arrive to the system and are stored into a single job queue. We suppose that a job j may be sequential or multi-threaded, that jobs are allocated to a machine according to the space sharing policy, and that each job is independent, i.e the execution of a job does not depend on the execution or results of others jobs. We also assume that all jobs are not preemptable, and that mechanisms to notify configuration changes, such as job submission/ending, are available in the computing platform. The computing grid is a dedicated one composed by many independent clusters of heterogeneous processing nodes, which can be a single-processor and/or SMP machines. Each cluster includes machines located at a specific site. Links among machines belonging to the same cluster are low-latency and high-bandwidth, while clusters are connected by the internet infrastructure. Furthermore, we suppose to have a set L of available software licenses (referred as license/s in the rest of the paper), with each license $l \in L$ executable on each machine present in the system, and that, at any time, the number of active licenses must not be greater than their availability. Each job can require a subset $l_c \subseteq L$ of licenses to be executed.

Submitted jobs, machines and licenses are annotated with information describing computational requirements and hardware/software features, respectively. Each job is described by an identifier, a submission and deadline time, an estimation of its duration, a benchmark score, which represents the computational power of the machine used for the time estimation, and the number and type of processors and licenses requested. Machines are described by a benchmark score, the number and type of CPUs and licenses they can run. Each license is described by a type and its availability. The scheduler aims to schedule arriving jobs respecting their deadlines and user peculiarities, and optimizing license and machine usage.

4 The two-level scheduler architecture

Figure 1 shows the proposed two-level scheduler architecture. At the highest level there is the *Meta-Scheduler* (MS), which manages the queue to which users submit their jobs, and makes decisions to dispatch jobs to the lower-level scheduling instances, the *Local Schedulers* (LSs). A LS manages a job queue local to a cluster. It makes decisions to schedule jobs on the cluster machines. MS and LS classify incoming jobs according to two different sets of heuristics, in which each heuristics manages a specific problem constraint (e.g. deadline, requested licenses, user peculiarities). To each heuristics is associated a weight, which fixes the importance of the heuristics according to the system management policies adopted by an installation. The classification phase at both the levels exploits the same job attributes, because they are representative of the job QoS requirements. MS computes job priorities only using information describing the submitted jobs. Afterwards, jobs are dispatched to LSs according to the MS's classification and the workload on each cluster. At LS level, the computation of job priorities also exploits dynamic information about the status of hardware/software resources, and job queued times. As a result, the complexity of the heuristics at LS level is greater than those used at MS level. According to the LS's classification and the hardware/software resource usage, queued job are scheduled to the available computational resources.

Job priorities are computed at LS level at each new scheduling event, which is job queued or ending. In the considered context, since new jobs can be submitted to the system at any time, the job stream length changes dynamically. Therefore, in order to meet job deadlines and to avoid wasting resources, fast scheduling decisions are needed to prevent computational resources from remaining idle, and scheduling decisions from aging. Moreover, computing the priority value at each new scheduling event allows to better meet the scheduler goals [3].

Meta-Scheduler. The MS dispatches jobs to clusters according to a policy based on two functions: *Load* and *Ordering*. Load aims to dispatch jobs among clusters bal-

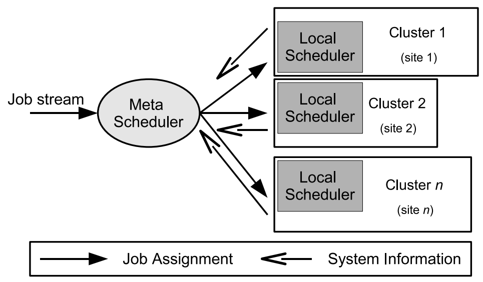

Fig. 1 Structure of the two-level scheduler architecture.

ancing the workload by assigning a job to the less loaded cluster. The workload on a cluster is estimated by summing the load due to the jobs queued to it. The load due to a job is defined as its estimated execution time multiplied by the benchmark value of the machine on which this time is computed. Ordering aims 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 higher priority jobs can be executed before the lower priority ones. 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 is defined for each one. *max* is the number of possible priority values that MS can assign to submitted jobs. 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 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. When a job is assignable to some eligible clusters, the problem is to find the cluster that can run it as soon as possible, also improving the number of jobs that are executed respecting their QoS requirements. Supposing to have a job j with priority P. Load uses P to access the cluster arrays. The first found cluster with 0 in the *P-th* array position is the chosen one for the job assignment. Indeed, according to the above 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-th* position of each array is greater than 0, the job is queued to the LS whose array stores in that position the lowest value. It will be selected to be scheduled later, when the execution of higher priority jobs, and of jobs with the same priority, but early arrived, is completed.

When there are more clusters with the same value in the *P-th* position, Load cannot be enough to guarantee that jobs are executed by respecting their QoS requirements. Randomly choosing a cluster in such subset, it can happen that a number of jobs with the same priority could be assigned to only few clusters. This way could lead slower priority jobs to be executed before than higher priority ones. To balance the number of jobs with equal priority in each cluster queue, the Ordering function is exploited. 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. When there are more clusters with the same number of jobs with equal priority in the *P-th* position, the one to dispatch a job is randomly selected.

In order to classify submitted jobs, at this level we exploit three heuristics: *Dead-Line*, *Licenses* and *User*, with each one managing an aspect of the considered problem. The job priority is computed summing the partial priority values $\Delta_{p,j}$ computed by each heuristics.

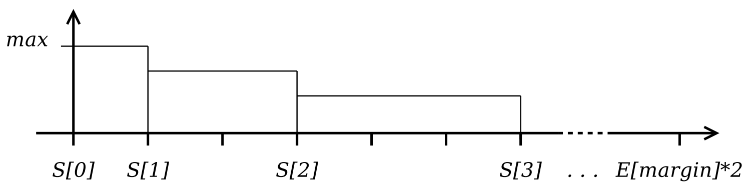

Fig. 2 Graphical representation of the $[0, E[margin] * 2]$ interval subdivision.

Deadline aims 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. Deadline requires an estimation of the job execution time in order to evaluate its completion time with respect to the current wall-clock time. It evaluates how much a job is "faraway" from the time at which it has to start its execution to respect its deadline. Its contribution $\Delta_{p,j}$ to the priority of a job j is proportional to the "proximity" of the time at which it has to start its execution to meet its deadline. Since MS has no knowledge about resources, Deadline classifies jobs as a function of information describing the job computational requirements.

Let $margin_j$ be the difference between the time at which j must start its execution in order to respect its deadline, and its submission time. The average margin value $E[margin]$ is computed as: $E[margin] = (\sum_{j=1}^{N} margin_j)/N$, where N is the window size, which specifies the number of jobs analyzed before j, and contributes to compute $E[margin]$. A job j is considered a "proximity" one if $margin_j < E[margin]$, otherwise it is considered to be a "faraway" one. To compute $\Delta_{p,j}$, Deadline 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 $p-1$. The use of $E[margin] * 2$ permits us to overestimate the time interval in which a job is a "proximity" one, while the exponential priority distribution permits us 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 finer granularity in the job priority process assignment.

The following expression formalizes the priority distribution we used to configure the Deadline heuristics. Let s and z be two integers representing two different priority values, with $s < z$, let $\#job_z$ be the number of jobs with priority value equal to z, then Deadline behaves at the best when:

$$\#job_z = \frac{\#job_s}{2^{z-s}}$$

In order to assign priority values to jobs using their $margin$ values, the interval $[0, E[margin] * 2]$ is divided into s as depicted in Figure 2. The subintervals $interval$ are computed as: $interval_{(max-k)} = [S_k, S_{k+1}]$ with $k = 0, ..., max - 1$, where max is the highest priority value a job can assume, and S_k is computed as:

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

where $minunity$ is given by: $minunity = (2 * E[margin]) / \sum_{k=1}^{max} 2^k$.

Finally, $\Delta_{p,j}$ is fixed equal to $(max-k)$, i.e. equal to the index of *interval* to which the job *margin* value belongs to. In this way, $\Delta_{p,j}$ is computed not considering the estimated job execution time. As a result, jobs with both large estimated execution time and large *margin* value obtain lower priorities than jobs with small estimated execution time and small *margin*. It can lead to execute large jobs not respecting their deadline. To face this aspect we refine the computation of $\Delta_{p,j}$. We consider the ratio r_j computed as: $r_j = margin_j/estimated_j$, where $estimated_j$ is the estimated execution time of j. $r_j \geq 1$ means that the job j can be classified as a "faraway" one, and, in this case, $\Delta_{p,j}$ is decreased as:

$$\begin{cases} \Delta_{p,j} = \Delta_{p,j} - \lfloor r_j \rfloor & \text{if } \ (\Delta_{p,j} - \lfloor r_j \rfloor) \geq 0 \\ \Delta_{p,j} = 0 & \text{if } \ (\Delta_{p,j} - \lfloor r_j \rfloor) < 0 \end{cases}$$

$r_j < 1$ means that the job j can be classified as a "proximity" one, and $\Delta_{p,j}$ is increased as follows. Let consider the interval $[0,1]$ divided into subintervals sub_w computed as: $sub_w = [\frac{1}{2^{w+1}}, \frac{1}{2^w}]$ with $w = 0, ..., max-1$, where w is the index of the subinterval. Then we identify to which sub_w r_i belongs to, and $\Delta_{p,j}$ is updated as:

$$\begin{cases} \Delta_{p,j} = \Delta_{p,j} + w & \text{if } \ (\Delta_{p,j} + w) \leq max \\ \Delta_{p,j} = max & \text{if } \ (\Delta_{p,j} + w) > max \end{cases}$$

The License heuristics computes $\Delta_{p,j}$ to favor the execution of jobs that improve the contention on the licenses usage. $\Delta_{p,j}$ is computed as a function of the number of licenses required by a job. Jobs asking for a high number of licenses get a boost in preference that gives them an advantage in scheduling. This pushes jobs using many licenses to be scheduled first, this way releasing a number of licenses.

Without lost of generality, at this level, we assume that all the grid machines are able to run all the available licenses. When licenses need to be considered for job assignment to individual clusters, a simple filter could be used to select the subset of machines, and the related clusters, that support the execution of the licenses requested by a job.

To compute $\Delta_{p,j}$ the heuristics considers the number $|L|$ of different kind of licenses available in the system. The interval $[0, |L|]$ is divided in max subintervals of the same size. Each subinterval corresponds to a priority value (e.g. at the first subinterval corresponds to the lowest priority value). $\Delta_{p,j}$ is computed as a function of the number of licenses requested by a job, and it is fixed equal to the related subinterval index.

The User heuristics computes $\Delta_{p,j}$ in order to execute a job j 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, such as user importance, kind of resources/services requested, kind of project.

Local-Scheduler. As local scheduler we adopted a Flexible Backfilling algorithm that selects the machines suitable to perform a job considering the number of processors and the licenses exploitable on a machine. At this level, the priority value assigned to each job j is the weighted sum of the contributions $\Delta_{p,j}$ computed by four heuristics: *Aging*, *Deadline*, *Licenses*, and *Wait Minimization*, which are obtained from the ones proposed in [3]. Job priorities may be recomputed at each new scheduling event, which is job queued or ending.

The goal of Aging is to avoid job starvation. It computes $\Delta_{p,j}$ as a function of the age that a job reached in the system, higher scores are assigned to jobs queued for a longer time. $\Delta_{p,j}$ of a job j is computed as:

$$\Delta_{p,j} = W_A \cdot (wallclock - submit_j)$$

where W_A is the heuristics weight, *wallclock* is the value of the system wall-clock at which the heuristics is computed, and $submit_j$ is the time at which the job j was submitted to the scheduler.

The Deadline heuristics aims to maximize the number of jobs that terminate their execution within their deadline. It requires an estimation of the job execution time.

The heuristics assigns 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 *max*, the score assigned to the job is increased in inverse proportion with respect to such distance. The *min* and *max* values are defined by system administrators. They identify the weight of the heuristics. The $\Delta_{p,j}$ value is set to *min* for jobs exceeding their deadline before to be scheduled.

A job is scheduled on the first found eligible machine with the highest computing power. Since jobs with a closer deadline receive higher priority, this strategy should improve the number of jobs executed within their deadline. Let $estimated_j$ and $deadline_j$ be the estimated execution time and the deadline of job j, respectively. Let us define:

$$\begin{aligned} nxt_j &= estimated_j \cdot \frac{bm_{\bar{m}}}{bm_m} \\ ext_j &= wallclock + nxt_j \\ t_j &= deadline_j - k \cdot nxt_j \end{aligned}$$

where bm_m is the most powerful cluster machine m (optimistic prediction), and $bm_{\bar{m}}$ is the power of the machine $\bar{m}$ used to estimate the execution time of j, nxt_j denotes the job's estimated execution time, ext_j denotes the estimated termination time of the job with respect to the current wall-clock time, and t_j is the time from which the job must be evaluated to meet its deadline (i.e. the job priority is updated also considering its deadline). k is a constant value fixed by system administrator, which permits to over estimate nxt_j. $\Delta_{p,j}$ is computed as:

$$\Delta_{p,j} = \begin{cases} min & \text{if } extime_j \leq t_j \\ a(ext_j - t_j) + min & \text{if } \gamma = true \\ min & \text{if } ext_j > deadline_j \end{cases}$$

where a is the angular coefficient of the straight line passing through the points (t_j, min), and $(deadline_j, max)$ and γ is true if $t_j < ext_j \leq deadline_j$.

The Licenses heuristics computes $\Delta_{p,j}$ to favor the execution of jobs that increase the critical degree of licenses. The rationale is that when these jobs end their execution, a set of licenses may become non critical, and the scheduler is able to take more flexible scheduling choices. A license becomes critical when there is a number of requests greater than the available number of its copies. The priority updating value is computed as a function of the licenses, critical and not critical, requested by a job. It assigns a higher score to jobs requiring a larger number of critical licenses. Let us define:

$$\begin{aligned}
\rho_l &= requests_l / total_l \\
l_{c,j} &= l \in licenses\ required\ by\ j | \rho_l > 1 \\
l_{\bar{c},j} &= l \in licenses\ required\ by\ j | \rho_l \leq 1
\end{aligned}$$

where $requests_l$ specifies how many jobs are requesting the license l, $total_l$ specifies how many copies of l can be simultaneously active, and a license is considered critical if $\rho_l > 1$. Δ_{p_j} is computed as:

$$\Delta_{p,j} = W_L \cdot (\sum_{l \in l_{\bar{c},j}} \rho_l + d \cdot \sum_{l \in l_{c,j}} \rho_l)$$

where W_L is the heuristics weight, and $d = max|\bigcup_{\forall j} l_{\bar{c},j}|, 1$.

The Wait Minimization heuristics aims to favor jobs with a shorter estimated execution time. The rationale is that shorter jobs are executed as soon as possible in order to release the resources they have reserved, and to improve the average waiting time of the jobs in the scheduling queue. $\Delta_{p,j}$ is computed as:

$$\Delta_{p,j} = W_{WM} \cdot \frac{minext_j}{estimated_j}$$

where W_{WM} is the heuristics weight, and $minext_j = min(estimated_j \mid j \in NJQ)$, with NJQ set of jobs queued to a cluster.

5 Performance Evaluation

In this section we present the evaluation of the scheduling solution carried out by the proposed two-level scheduler. The objective is to investigate the feasibility of the scheduling policies and the job classifications we propose. To this end, experiments were conducted by using the event-based simulator described in [3]. Three different cases were evaluated:

1. MS Heuristics: MS classifies submitted jobs and dispatches them to LSs. At LS level, scheduling decision are made by means of a Flexible Backfilling algorithm, which exploits job priorities computed by MS. Any job prioritization is performed at LS level. Higher the job priority is, higher the position of the job in LSs' queues is.
2. LS Heuristics: MS classifies submitted jobs and dispatches them to LSs, which prioritize incoming job. LSs recompute job priorities at each scheduling event (submission/ending of a job). This introduce a computational cost not present in the previous case. LS queued jobs are scheduled by means of a Flexible Backfilling algorithm.

3. NO Heuristics: 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.

The quality of the schedules computed by the proposed scheduler was evaluated by using the following metrics:

– Percentage of workload elaborated by each cluster. It shows the ability of the policies adopted at MS level to dispatch jobs by balancing the workload among clusters.
– Percentage of jobs executed not respecting their deadline. It shows the ability of the proposed solution to schedule jobs in such a way it maximizes the number of jobs executed respecting this QoS requirements.
– Percentage of system and license usage. It shows how the job classification and scheduling solutions adopted both at MS and LS level allow a fruitful exploitation of the available processors and licenses.
– Average slowdown of jobs without deadline. It shows how the system load delays the execution of such jobs. It is computed as: $AverageSlowdown(j) = (Tw_j + Te_j/Te_j)/\#\ of\ processed\ jobs$, where Tw_j and Te_j are the waiting and the execution time of the job j, respectively. Closer the $AverageSlowdown$ to 1, greater the ability of the scheduler to avoid the job queues increasing is.

The evaluation was conducted by using four different streams of 5000 jobs generated according to a negative exponential distribution with jobs inter-arrival time Ta equal to 0, 5, 10, and 15 simulator time unit. It permits us to evaluate the behavior of MS when the job submission rate increases. We simulated a grid composed by 225 machines, distributed on four clusters, called Cluster1, Cluster2, Cluster3 and Cluster4, each one including 120, 60, 30, and 15 machines, respectively. To each license we associate the parameter $licenses\_ratio$ that specifies its maximum number of copies concurrently usable, expressed as the percentage of the number of machines able to run it. For each simulation, we randomly generated a list of jobs and machines whose parameters are generated according to a uniform distribution in the ranges: $estimated[8000 - 10000]$, $benchmark[100 - 500]$, $margin[1500 - 5500]$, $CPU[4 - 32]$, $licenses\_ratio[50\% - 70\%]$. The benchmark and CPU parameters are chosen for both jobs and machines. Moreover, in each simulation, 30% was the probability that a job needs a license, and the 30% of jobs were generated without deadline. The deadline of a job j is generated according to the following expression: $deadline_j = submission_j + estimated_j + margin_j$. 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. Each test was conducted by randomly generating a list of machines, and, in order to obtain stable values, each simulation was repeated 20 times with different job parameter values.

Figure 4 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. It is the main metric we used to prove that MS is able to dispatch jobs among the underlying clusters balancing the workload. 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 the kind of jobs queued to a cluster. Greater the number of jobs of different kind is, greater the capability of Backfilling to efficiently exploit the

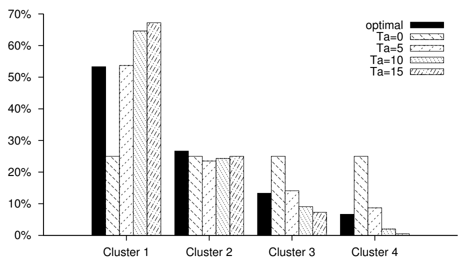

Fig. 3 Average clusters load.

underling 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 4) 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 their execution starts. Since MS dispatches 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 maintain 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 MS job classification, we show the results obtained by using $Ta = 5$ concerning the other evaluation metrics used.

Figure 4 (left) shows the average slowdown evaluated for each cluster considering jobs without deadline. It improves in inverse proportion to the cluster computational power. Since MS dispatches jobs trying to have the same workload in each cluster queue, the average slowdown of jobs submitted without deadline grows when the cluster computational power decreases. Moreover, the average slowdown is higher than 1 in each cluster through simulations. This means that LSs have sufficient jobs to process, therefore the job inter-arrival time chosen is suitable for the purpose of our evaluation.

Figure 4 (right) shows the percentage of jobs executed do not respecting their deadlines. MS Heuristics and LS Heuristics are able to improve the number of jobs executed within their deadline comparing with No Heuristics, which exploits the FCFS order. MS Heuristics carries out results comparable to those obtained by LS Heuristics, but with a smaller computational cost. Furthermore, considering the job distribution performed by MS, in the case of Cluster 1, hundreds of jobs are executed missing their deadlines, while in the case of Cluster 4 they are only few.

Figure 5 shows the percentage of system (left) and license (right) usage (expressed in percentage) , respectively. These values are computed according to the following expression:

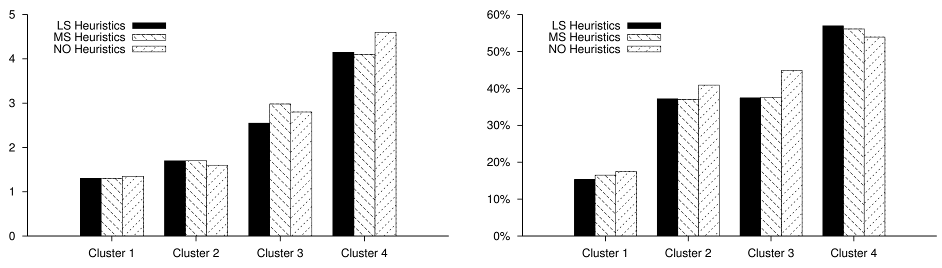

Fig. 4 Average slowdown of jobs without deadline (left), percentage of jobs executed do not respecting their deadline (right).

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

where *res* means "processors" or "licenses" when system or license usage is computed, respectively.

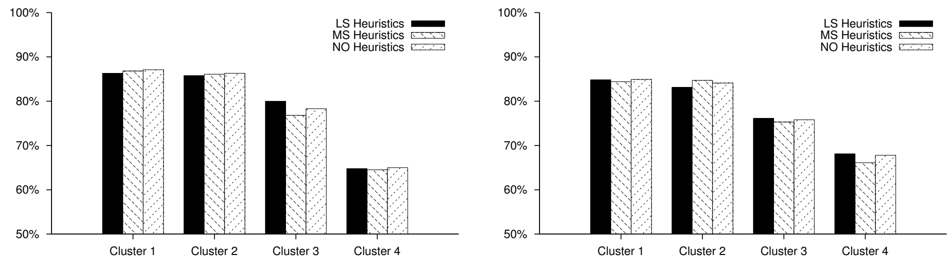

Fig. 5 Average percentage of processor (left) and license (right) usage.

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

6 Conclusion and future work

This paper describes the study we conducted to develop a two-level scheduler to dynamically schedule a continuous stream of batch jobs on large-scale grids made up of heterogeneous machines in interconnected clusters. The proposed solution aims to schedule arriving jobs balancing the clusters workload, respecting the job running requirements and deadlines, and optimizing the utilization of hardware and software resources. The proposed solution exploits a set of heuristics, each one managing a specific problem constraint, that guide the Meta and Local schedulers in making scheduling decisions. We investigated two solutions that use different heuristics to compute jobs priorities. One exploited at MS level, which computes the job priorities at job submission time, the other one, exploited at LS level, which re-computes the job priorities at every scheduling event (submission/ending of a job). The conducted simulation tests demonstrated that the investigated solution can be a viable one. In particular, we show that using a lightweight component like MS joined with lightening LSs, carries out good results as using more complex LSs. The proposed solution could be extended identifying possible heuristics refinements and extensions to enhance the current scheduler to support different local scheduler policy. Moreover, to completely understand the potentiality of the proposed solution, it should be interesting to evaluate it by using log file describing the execution of real jobs.

Acknowledgements 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. VIOLA, Vertically Integrated Optical Testbed for Large Application in DFN (2005)
2. Buyya, R., Abramson, D., Giddy, J.: 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)
3. Capannini, G., Baraglia, R., Puppin, D., Ricci, L., Pasquali, M.: A job scheduling framework for large computing farms. In: Proceedings of SC07. Reno, USA (2007)
4. Capit, N., Costa, G.D., Georgiou, Y., Huard, G., n, C.M., Mouni, G., Neyron, P., Richard, O.: A batch scheduler with high level components. In: Proceedings of CCGrid05 (2005)
5. Casavant, T.L., Kuhl, J.G.: A taxonomy of scheduling in general-purpose distributed computing systems. IEEE Trans. Softw. Eng. **14**(2), 141–154 (1988)
6. Feitelson, D.D., Rudolph, L., Schwiegelshohn, U.: Parallel job scheduling, a status report. In: Lect. Notes Comput. Sci. vol. 3277, pp. 1–16. Springer-Verlag, London, UK (2004)
7. Feldmann, A., Sgall, J., Teng, S.H.: Dynamic scheduling on parallel machines. Theor. Comput. Sci. **130**(1), 49–72 (1994)
8. Fiat, A., Woeginger, G. (eds.): Online Algorithms, The State of the Art. Lecture Notes in Computer Science. Springer (1998)
9. Fiat, A., Woeginger, G.J.: Online algorithms, the state of the art. In: Developments from a June 1996 seminar on Online algorithms. Springer-Verlag, London, UK (Lect. Notes Comput. Sci. vol. 1442, 1998)
10. Garey, M.R., Graham, R.L.: Bounds for multiprocessor scheduling with resource constraints. SIAM J. Comput. **4**(2), 187–200 (1975)
11. Keller, Hovestadt, M., Kao, O., Streit, A.: Job scheduling strategies for parallel processing. In: In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, 9th International Workshop, JSSPP. Springer-Verlag, Seattle, USA (June 24, 2003)
12. Krallmann, J., Schwiegelshohn, U., Yahyapour, R.: On the design and evaluation of job scheduling algorithms. In: Proceedings of IPPS/SPDP '99/JSSPP '99, pp. 17–42. Springer-Verlag, London, UK (1999)
13. Mohamed, H.H., Epema, D.H.J.: Experiences with the koala co-allocating scheduler in multiclusters. In: Proceedings of CCGRID '05, pp. 784–791. IEEE Computer Society, Washington, DC, USA (2005)
14. Muálem, A.W., Feitelson, D.G.: Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. IEEE Trans. Parallel and Distributed Syst. **12**(6), 529–543 (2001)
15. Noël, S., Delannoy, O., Emad, N., Manneback, P., Petiton, S.G.: A multi-level scheduler for the grid computing yml framework. In: Euro-Par Workshops, pp. 87–100 (2006)
16. Schwiegelshohn, U., Yahyapour, R.: Analysis of first-come-first-serve parallel job scheduling. In: SODA '98: Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms, pp. 629–638. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (1998)

17. Wldrich, O., Ziegler, W., Wieder, P.: A meta-scheduling service for co-allocating arbitrary types of resources. Tech. Rep. TR-0010, Institute on Resource Management and Scheduling, CoreGRID - Network of Excellence (2005). URL http://www.coregrid.net/mambo/images/stories/TechnicalReports/tr-0010.pdf