A Job Scheduling Framework for Large Computing Farms

Gabriele Capannini
Information Science and Technologies Institute
Via G. Moruzzi 1
56126 Pisa, Italy
gabriele.capannini@isti.cnr.it

Ranieri Baraglia
Information Science and Technologies Institute
Via G. Moruzzi 1
56126 Pisa, Italy
ranieri.baraglia@isti-.cnr.it

Diego Puppin
Information Science and Technologies Institute
Via G. Moruzzi 1
56126 Pisa, Italy
diego.puppin@isti.cnr.it

Laura Ricci
Department of Computer Science
Largo B.Pontecorvo
56100 Pisa, Italy
ricci@di.unipi.it

Marco Pasquali *
Information Science and Technologies Institute
Via G. Moruzzi 1
56126 Pisa, Italy
marco.pasquali@isti.cnr.it

ABSTRACT

In this paper, we propose a new method, called Convergent Scheduling, for scheduling a continuous stream of batch jobs on the machines of large-scale computing farms. This method exploits a set of heuristics that guide the scheduler in making decisions. Each heuristics manages a specific problem constraint, and contributes to carry out a value that measures the degree of matching between a job and a machine. Scheduling choices are taken to meet the QoS requested by the submitted jobs, and optimizing the usage of hardware and software resources. We compared it with some of the most common job scheduling algorithms, i.e. Backfilling, and Earliest Deadline First. Convergent Scheduling is able to compute good assignments, while being a simple and modular algorithm.

Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems–Sequencing and scheduling

General Terms

Algorithms, Theory, Experimentation, Performance

Keywords

Job scheduling, Deadline Scheduling, Software License Scheduling, Computing Farm

*Institutions Markets Technologies, Institute for Advanced Studies, Piazza S. Ponziano, 6, 55100 Lucca, Italy.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SC07 November 10-16, 2007, Reno, Nevada, USA
(c) 2007 ACM 978-1-59593-764-3/07/0011...$5.00

1. INTRODUCTION

Computing farms provide the ability to harness the power of a set of computational resources as an ensemble. A computing farm can integrate heterogeneous hardware and software resources such as workstations, parallel systems, servers, storage arrays, and software licenses (referred as license/s in the rest of the paper). In such an environment, users should submit their computational requests without necessarily knowing on which computational resources these will be executed. In the past, a lot of research effort has been devoted to develop job scheduling algorithms. Nowadays, many of these algorithms are exploited in commercial and open source job schedulers. However, none of these scheduler strategies are able to deal with the entire range of constraints (e.g. number of licenses concurrently usable) and requirements (e.g. jobs' deadline) presented by the users, and analyzed in this work.

In this paper, we propose a framework, to schedule a continuous stream of batch jobs on the machines of large computing platforms, made up of heterogeneous, single-processor or SMP nodes, linked by a low-latency, high-bandwidth network. The proposed solution is based on the Convergent Scheduling technique [17], which 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 requirements, and optimizing the utilization of hardware resources as well as software resources.

The proposed solution permits us to carry out an effective job scheduling plan, on the basis of the current status of the system (i.e. resource availability, executing jobs), and information related to jobs waiting for execution. In order to take decisions, the scheduler assigns priorities to all the jobs in the system, which are then used to determine a matching.

The rest of this paper is organized as follows. Section 2 describes some of the most common job scheduling algorithms.

Section 3 gives a description of the problem. Section 4 describes our solution. Section 6 outlines and evaluates the proposed job scheduler through simulation. Finally, conclusion and future work are described in Section 7.

2. RELATED WORK

A major effort has been devoted to understand job scheduling problems in scientific literature. Job scheduling algorithms can be divided into two classes: on-line and offline [9, 5]. On-line algorithms are those that do not have knowledge on the whole input job stream. They take decisions for each arriving job not knowing future input. Examples of on-line scheduling algorithms are: First-Come-First-Serve [10, 21], Backfilling [13, 7], List Scheduling [8, 18]. Offline algorithms know all the jobs before taking scheduling decisions. Examples of offline algorithms are: SMART [20], Preemptive Smith Ratio Scheduling [19], Longest Reference First [12]. In the following, we give a briefly description of the most common job scheduling algorithms. Many of these algorithms are now used in commercial and open source job schedulers [6], such as the Maui scheduler [11], Portable Batch System [2], and Sun Grid Engine [15]. However, none of these schedulers can deal with the entire range of constraints, such as number of licenses concurrently usable and jobs' deadline, we address in our work.

The *Backfilling* algorithm requires that each job specifies its estimated execution time, so that the scheduler can predict when jobs finish and other ones can be started. Jobs executing longer than their estimated duration are deleted. While the job at the head of the queue is waiting, the scheduler tries to schedule jobs that are lower in the queue, which can exploit idle resources without delaying the execution of the job at the top of the queue. This allows using processors that would otherwise remain idle. Executing some jobs out of order, other jobs may get delayed. To avoid job starvation, backfilling will never completely violate the First Come First Served (FCFS) order. 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) [16], Conservative, and Flexible backfilling [7].

In the original EASY backfilling algorithm, only the first queued job receives a reservation. 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. Other jobs can use resources that do not violate the reservation.

The Conservative variant tries to reduce the delays in the execution of the waiting jobs by making reservations for all jobs. However, simulation results [7] 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. More recently, [23] presents a backfilling solution that exploits adaptive reservations as a function of the job delay, due to previous backfilling decisions. If a job delay exceeds a predefined threshold, a reservation is made for it.

The Flexible variant was obtained from the original EASY backfilling by prioritizing jobs according to scheduler goals. Jobs are then queued according to their priority value, and selected for scheduling (including candidates for backfilling) according to this priority order [23, 3].

Other backfilling variants consider the amount of lookahead into the queue [22]. Here, in order to reduce the resource fragmentation effect, the whole queue is considered at once, in order to find the set of jobs that together maximize performance.

Earliest deadline first (EDF) is a dynamic scheduling method, which arranges jobs in a priority queue, and the next job to be selected is the one with highest priority. The priority of a job is inversely proportional to its deadline, and it can change during its execution. EDF is able to carry out an optimal scheduling algorithm on preemptive uniprocessors, while, when the system is overloaded, the set of processes that will miss deadlines is largely unpredictable [1]. EDF is often used in real-time operating systems.

3. PROBLEM DESCRIPTION

In our study, the set of batch jobs (i.e. queued + running) and the machines in the system are enumerated by the sets N and M, respectively. We assume that a job $i \in N$ is sequential or multi-threaded, and that each machine $m \in M$ is executing only a single job at a time. We also assume that all jobs are preemptable. A job's execution may be interrupted whenever a schedule event (job submission or ending) happens. The building of a scheduling plan involves all the N jobs in the system. The scheduler aims to schedule arriving jobs respecting their deadline requirements, and optimizing license and machine usage.

Submitted jobs and machines are annotated with information describing computational requirements and hardware/-software features, respectively. Each job is described by an identifier, its deadline, an estimation of its duration, a benchmark score, which represents the architecture used for the time estimation, the number and type of processors and licenses requested, and the amount of input data. Machines are described by a benchmark score, the number and type of CPUs and licenses they can run.

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.

4. CONVERGENT SCHEDULING FOR BATCH JOB SCHEDULING

In order to exploit the convergent scheduling method, we define a matrix $P^{|N| \times |M|}$, called Job-Machine matrix. Each matrix entry stores a priority value specifying the degree of preference of a job for a machine. As shown in Figure 1, our scheduling framework is structured according to three main phases: Heuristics, Clustering & Pre-Matching, and Matching.

To manage the problem constraints, we define several heuris-

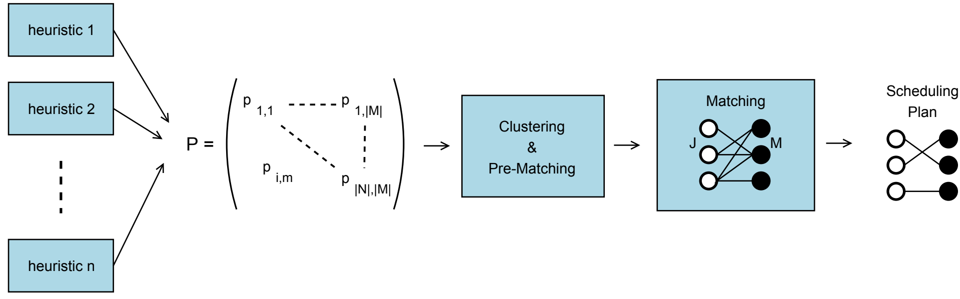

Figure 1: Structure of the Convergent Scheduling framework.

tics that guide the scheduler in making decisions. The Job-Machine matrix constitutes the common interface to heuristics. Each heuristics changes priority values in order to compute jobs-machines matching degrees.

Managing the constraints by using distinct heuristics, and structuring the framework in subsequent phases, leads to a modular structure that makes it easier to extend and/or to modify.

In our case, $|N|$ changes dynamically, as new jobs can be 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.

The Clustering & Pre-Matching phase aims at removing conflicts on the license usage. It first clusters jobs according to their requests for licenses. Then, the Multidimensional 0-1 Knapsack Problem algorithm (MKP) [14] is applied to the jobs composing each cluster to find the subset of jobs that can be simultaneously executed without violating the constraints on the license usage. The jobs in the remaining subset will be discarded, i.e. their entries are "cleared" from the Job-Machine matrix. Discarded jobs are then reconsidered to build the new scheduling plan, after the next scheduling event.

The Matching phase elaborates the resulting P matrix to carry out the job-machine associations, i.e. 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 associations $job - machine$ to which correspond a larger preference degree, according to the problem constraints.

4.1 Collection of heuristics

Seven heuristics have been implemented to build our Convergent Scheduling framework. The order of magnitude of the heuristics complexity is equal to $(|N| \cdot |M|)$.

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 floating licenses (i.e. number and type).

Deadline. The aim of the Deadline heuristics is to compute the Job-Machine update value $\Delta_{p_{i,m}}$ in order to execute jobs respecting their deadline. Jobs closer to their deadline get a boost in preference that gives them an advantage in scheduling. In the Job-Machine matrix, the priorities values $p_{i,m}$ on rows corresponding to "late" jobs are increased proportionally to the proximity of their deadline. For each job $i \in N$ the Δ values are computed w.r.t. the machines $m \in M_{Aff_i}$ according to the following function:

$$f_1 = \begin{cases} Min & \text{if } Extime_{i,m} \leq t_{i,m} \\ \Gamma & \text{if } t_{i,m} < Extime_{i,m} \leq Deadline_{i,m} \\ Min & \text{if } Extime_{i,m} > Deadline_{i,m} \end{cases}$$

where Γ is computed as:

$$\Gamma = a(Extime_{i,m} - t_{i,m}) + Min$$

where a is the angular coefficient of the straight line passing through the points $(t_{i,m}, Min)$, and $(deadline_{i,m}, Max)$ (see Figure 2, which gives a graphical representation of f_1). Min is constant value fixed by the installation. $t_{i,m}$ 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). $t_{i,m}$ and $Extime_{i,m}$ are computed as follows:

$$\begin{aligned} t_{i,m} &= Deadline_{i,m} - k \cdot Nxtime_{i,m} \\ Nxtime_{i,m} &= (Estimated_i \times (1 - Progress_i)) \cdot \frac{BM_{\bar{m}}}{BM_m} \end{aligned}$$

where $Progress_i$ is the percentage of i already executed, BM_m is the power of the machine m where i is executing, and $BM_{\bar{m}}$ is the power of the machine m utilized to estimate the execution time of i. This time can be evaluated statistically by analyzing historical data or by benchmarking. k is a constant value fixed by the installation. It permits us to overestimate $Nxtime_{i,m}$, i.e. the time from which f_1 starts the increasing phase. With $k = 1$, i.e. without overestimation, any event able to delay the execution of a job would lead to a violation of its deadline. Fixing $k = 2$, f_1 will lead to update the Job-Machine matrix entries not starting from

the last available time (i.e. $Deadline_{i,m} - Nxtime_{i,m}$) but starting from twice the value $Nxtime_{i,m}$.

$Extime_{i,m}$ is the time at which a job is estimated to end its execution. It is computed as:

$$Extime_{i,m} = Now + Nxtime_{i,m}$$

where Now is the current wall clock time.

The values to update the Job-Machine entries are computed according to the following expressions:

$$\Delta_{p_{i,m}} = \left(\sum_{k=1}^{|M_{Aff_i}|} f_1(Extime_{i,k}) \right) \cdot \frac{\bar{f}_1(Extime_{i,m})}{d_i} \quad (1)$$

$$\bar{f}_1(Extime_{i,m}) = Max - f_1(Extime_{i,m})$$

$$d_i = \sum_{m=1}^{|M_{Aff_i}|} \bar{f}_1(Extime_{i,m})$$

The first term of (1) establishes the job "urgency" to be executed respecting its deadline. A higher value of $f_1(Extime_{i,m})$ for a job means that it is a candidate (i.e $Extime_{i,m}$ is very close to $Deadline_{i,m}$) to finish its execution violating its deadline when executed on m. High values of the first term mean that the related job could be "late" on all, or almost all, the available machines. Therefore, the priority of such jobs has to be increased more than those of jobs that have obtained lower values. The second term of (1) distributes the values computed by the first term on the available machines, proportionally to the power of each machine. For each job, $\bar{f}_1(Extime_{i,m})$ establishes a ordering of the machines, w.r.t. their power, on which it can be performed. The higher the $\bar{f}_1(Extime_{i,m})$ value associated to a machine is, the higher the probability that such machine will perform the associated job respecting its deadline is. Max is a constant fixed by the installation (see Figure 2).

Licenses. This heuristics updates the job priorities to favour 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 licenses, critical and not critical, 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:

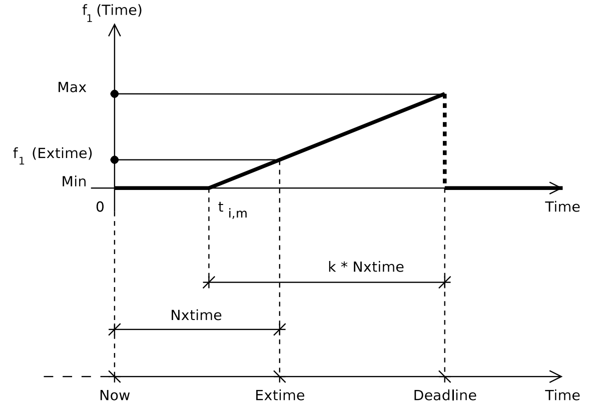

Figure 2: Graphical representation of f_1 with $k = 2$ and $Min = 0$.

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

where the numerator specifies the "request", i.e. how many jobs are requesting a license, and the denominator 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 $L_c = \{l \in L \mid \rho_l > 1\}$, and $L_{nc} = L \setminus L_c$, $w()$ is defined as follow:

$$w(i) = |L| \cdot \sum_{l=1}^{|L_c|} s_{i,l} \cdot \rho_l + \sum_{l=1}^{|L_{nc}|} s_{i,l} \cdot \rho_l \quad \forall i \in N$$

where $|L|$ is used as a multiplicative constant to give more weight to critical licenses.

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

$$w_1(i) = \sum_{l=1}^{|L_c|} s_{i,l} \cdot \rho_l \quad w_2(i) = \sum_{l=1}^{|L_{nc}|} s_{i,l} \cdot \rho_l$$

$$a < b \text{ iff } \begin{cases} w_1(a) < w_1(b) & \text{with } w_1(a) \neq w_1(b) \\ w_2(a) < w_2(b) & otherwise \end{cases}$$

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

$$\Delta_{p_{i,m}} = LI \cdot \left(\frac{w(i)}{max(w(i))} \right) \quad \forall i \in N$$

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

Input. The goal of this heuristics is to update the matrix entries according to the cost due to the transfer of the job

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

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

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

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

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

where IN is a value that sets the importance of the heuristics.

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

$$\Delta_{p_{i,m}} = WM \cdot \left(1 - \frac{Remaining_{i,m}}{Sup_{Remaining}}\right)$$

where WM is a value, which sets the weight of the heuristics, $Remaining_{i,m}$ is computed according to the expression below, and $Sup_{Remaining}$ is set equal to the maximum value of $Remaining_i$.

$$Remaining_{i,m} = Estimated_{i,\bar{m}} \cdot (1 - Progress_i) \cdot \frac{BM_{\bar{m}}}{BM_m}$$

where $Estimated_{i,\bar{m}}$ is the estimation of the CPU time required to run a job i on $\bar{m}$, $Progress_i$ is the CPU time used by a job until the heuristics evaluation, $BM_{\bar{m}}$ is the power of the machine on which $Estimated_i$ has been valuated, and BM_m the power of a machine eligible for the execution of i.

Overhead minimization. Since a job preemption may happen at any schedule event, this heuristics aims to contain the overhead due to the interruption of a job execution on a machine and its resuming on another one, or on the same one at a different time. The heuristics tries to minimize job migrations by increasing of a predefined value Δ the priorities of the job-machine associations present in the previous scheduling plan.

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 reached in the system. It is function of the difference between the wall clock time and its submission time. $\Delta_{p_{i,m}}$ is computed as:

$$\Delta_{p_{i,m}} = AA \cdot (Wallclock - Submit_i) \ \ \forall m \in M_{Aff_i}$$

where AA is a constant used to set the importance of the heuristics.

In this work, we hand-tuned the main parameters of heuristics, but we plan to use an evolutionary framework to compute the best setting.

4.2 Clustering & Pre-Matching

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

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

The MKP (Multidimensional 0-1 Knapsack Problem) optimization method [14] is applied to each cluster to find the subset of jobs that could be simultaneously executed, without violating the constraint on license usage. The remaining subset will be discarded, i.e. their entries are "cleared" from the Job-Machine matrix. The resulting matrix will be passed to the Matching phase in order to carry out the job-machine associations, i.e. the new scheduling plan. In our case, the knapsack's capacities are the number of licenses of each specific type simultaneously usable, and the objects are the jobs that require such licenses.

Defining the costs c_i and the weights a_i of a job i as follows, the solution proposed by Eilon [14] can be exploited to formulate the MKP as a KP (0-1 Knapsack Problem).

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

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

To solve the resulting KP we choose a greedy algorithm with polynomial complexity, which adopts the decreasing unitary costs method as the ordering heuristics to select the objects, as follow.

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

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

5. MATCHING

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

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

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

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

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

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

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

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

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

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

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

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

IMSF is able to carry out a new scheduling plan exploiting partial results coming from the previous one. Instead of rebuilding the bipartite graph from scratch, it starts from a partial bipartite graph built exploiting the job-machine associations inherited by the previous matching phase. IMFS builds a new bipartite graph changing the way and the sign of the weight of the arcs, related to the inherited job-machine associations. Then, on the graph, the LPT is computed. On this tree we can distinguish two kind of paths: even and odd, w.r.t. the number of arcs visited. The first ones must end in a node representing a job, while the odd ones must end with a machine not yet associated to a job. When possible, odd paths are preferred to even paths because they lead to an improvement of the matching cardinality. Then, the path with the largest cost is selected, i.e. the one that leads to the largest increase of the matching size. The IMSF complexity is equal to $O(N^3)$, it is an approximation of MFS.

6. PERFORMANCE EVALUATION

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

In this work, all CS versions have used the same set of heuristics, the parameters of which were hand-tuned to give more importance to the job deadline requirement and the license usage constraint. We simulate job that are non preemptable by increasing the priority of the jobs being executed of a value (see Overhead minimization heuristics) higher than that used to increase the priority of the other jobs. In such a way, previous scheduling choices are preserved, and a scheduling plan is changed when a job ends. [1]

The evaluation was conducted by simulations using different streams of jobs generated according to a negative exponential distribution with different inter-arrival times between jobs. Moreover, each job and machine parameter was randomly generated from a uniform distribution.

For our evaluation, we developed an event-based simulator, where events are job arrival and termination. We simulated a computing farm, with varying number of jobs, machines and 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 $[500 - 3000]$.
- Deadline $[25 - 150]$.
- Number of CPUs $[1 - 8]$.
- Benchmark $[200 - 600]$.
- Input $[100 - 800]$.
- License ratio $[55\% - 65\%]$, percentage of the number of machines on which a license can be used.
- License suitability 90%.
- License needed by a job 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 Needed by a job" specifies the probability that a job needs a license.

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

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

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

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

To evaluate the quality of schedules computed by CS we exploited different criteria: the percentage of jobs that do not respect the deadline constraint, the percentage of machine usage, the percentage of license usage, and scheduling computational time. The evaluation was conducted by comparing our solution with EASY Backfilling (BF-easy), Flexible Backfilling, and Earliest Deadline First(EDF). Two different versions of the Flexible Backfilling algorithm, called BF-rm and BF-unrm, were implemented. BF-rm uses resource reservation for the first job in the submission queue, ordered according to a job priority value. Priority and reservation are updated at each job submission or ending event. BF-unrm maintains the first queued job reservation through events until the job execution. The job priority values are computed at the computation of a new scheduling plan using the CS heuristics. Computing the priority value at each new scheduling event permits us to better meet the scheduler goals [4].

To implement EDF, jobs were ordered in decreasing order w.r.t. their deadline, and machines were ordered in decreasing order w.r.t the value of the "Benchmark" attribute. Each job was scheduled to the best available machine, respecting the constraints on both the number of CPUs and of licenses. If an eligible machine is not found the job is queued and rescheduled at the next step of the scheduling process. It exploits the job preemptive feature to free resources to execute first new submitted job with an earliest deadline.

To evaluate the quality of the solutions carried out by the MKP algorithm, tests were conducted generating instances of job streams varying the value of the "License ratio" parameter in the ranges $[40\% - 45\%]$, $[45\% - 50\%]$, $[50\% - 55\%]$, $[55\% - 60\%]$, and fixing the job inter-arrival times at 10 sim-

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

ulator time unit. Table 1 shows the results obtained by running the branch&bound and MKP algorithms on the same job streams instances. For each instance and algorithm, the Table reports: the smallest (Sml), the best (Bst) and the average (Avg) percentage value of license usage, and the average algorithm execution time (Time).

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

To evaluate the schedules carried out by the CS schedulers we generated seven streams of jobs with jobs inter-arrival times (Ta in Figure 3) fixed equal to 4, 6, 8, 10, 12, 16 and 20 simulator time unit ($4 \div 20$ in Figures 4, 5, 6, 7). 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.

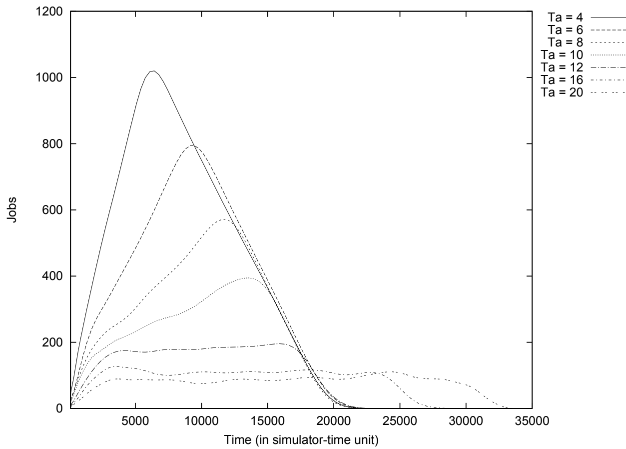

Figure 3: System workload through simulations.

We first measured the percentage of jobs executed that do not respect their deadline. Figure 4 shows this value for the seven job stream distributions. As expected, the smaller the job inter-arrival time is, the greater the job competition in the system is, and consequently the number of late jobs improves. Satisfactory results are obtained when the available cluster computational power is able to maintain almost constant the system contention. The three backfilling algorithms and EDF obtain worse results than those obtained by all the CS schedulers. Moreover, CS_ i_p and CS_ ni_p obtain equal or very close results demonstrating that the CS incremental version can be used without loss of results quality.

Figure 5 shows the slowdown parameter evaluated for jobs

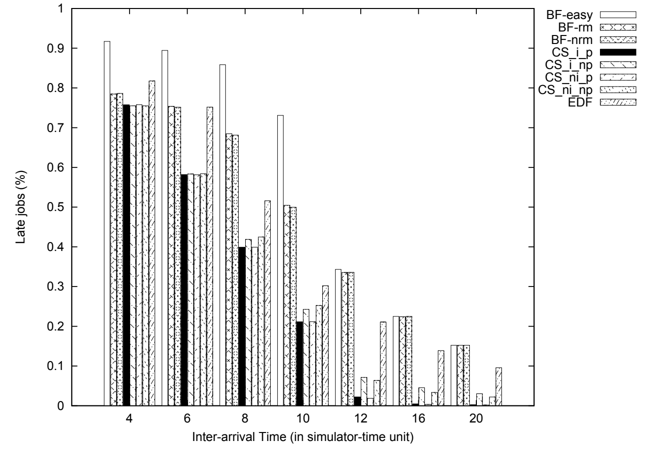

Figure 4: Percentage of jobs executed do not respecting their deadline.

without the deadline. It 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 [16]. This parameter gives us as the system load delays the execution of such jobs. As can be seen, for job streams generated with a inter-arrival time equal or greater than 10 simulator time unit, jobs scheduled by using a CS scheduler obtained a Tw equal or very close to 1.

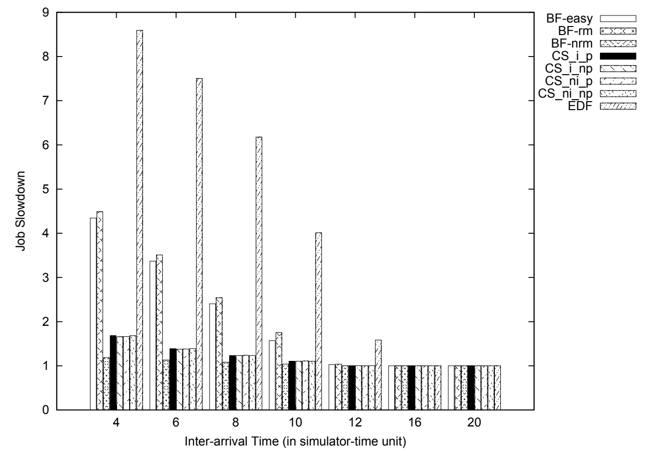

Figure 5: Slowdown of jobs submitted without deadline.

In Figure 6, the percentage of machine usage is shown. Since the considered algorithms do not support the processor space sharing allocation policy, this measure permits us to roughly figure out the system utilization. It is obtained averaging out the value:

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

computed at each instant of a simulation. A machine can remain idle when it does not support all the hardware/software

Table 1: Results obtained by running the algorithms branch&bound and MKP on the same job streams instances.

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

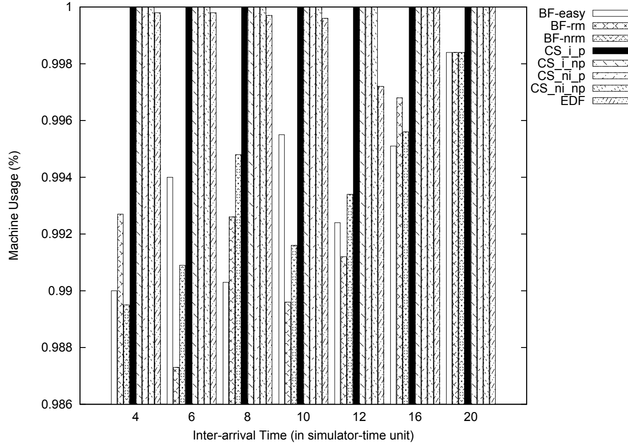

Figure 6: Percentage of machine usage.

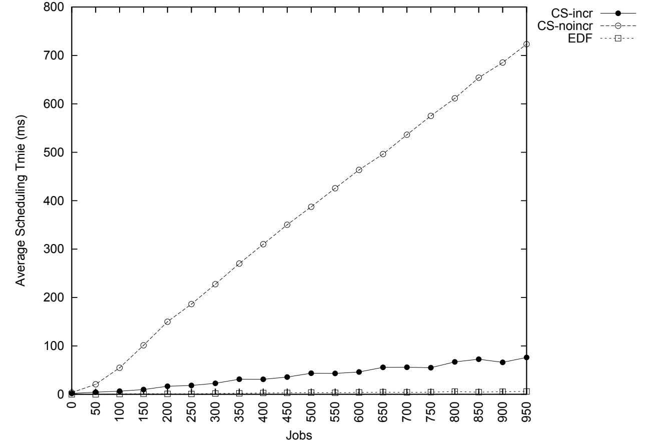

Figure 8: Algorithm scalability.

requirements required by a queued job. We can see the CS algorithms schedule jobs in a way that permits to maintain all the system machines always busy.

In Figure 7, we show the average scheduling times spent by the schedulers for conducting the tests on the simulated computational environment. As expected, the CS versions which support the preemption feature require more execution time. CS_ i_p and CS_ i_np obtain a lower execution time than CS_ ni_p and CS_ ni_np, respectively.

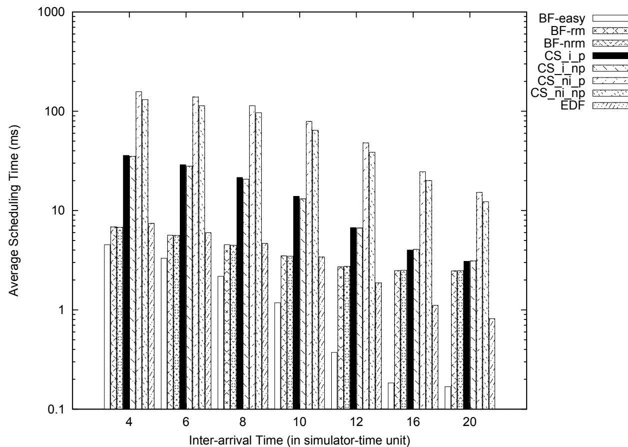

Figure 7: Average scheduling times.

In Figure 8 we show the scalability for the CS and EDF algorithms. The CS scheduler scalability was evaluated for the versions using the incremental approach (CS-incr) and non using the incremental approach (CS-noincr) both using the job preemption feature. It was evaluated measuring the time needed to carry out a new scheduling plan increasing the number of jobs. It can be seen that the CS-incr versions obtained very good scalability results.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we propose a new solution, based on the Convergent Scheduling technique, to dynamically schedule a continuous stream of batch jobs on large non-dedicated distributed and heterogeneous computing platforms. It aims at scheduling arriving jobs respecting their deadline requirements, and optimizing the exploitation of hardware and software resources. The proposed scheduler exploits a set of heuristics, each one managing a specific problem constraint, that guide the scheduler in making decisions. To this end the heuristics assign priorities to all the jobs in the system. The jobs priorities are re-computed at submission/ending of a job. A new scheduling plan is carried out on the basis of the current status of the system, and information related to jobs waiting for execution. The modular structure of the proposed scheduler makes simple to define a set of customized heuristics in order to meet the service goal of an installation.

The CS scheduler was evaluated by simulations using different streams of jobs with different inter-arrival times between jobs. Four different versions of the CS scheduler, which exploit different features, were evaluated comparing them with EDF and Backfilling schedulers. Even if the conducted evaluation constitutes the first step in analyzing the behavior

and quality of the presented approach, it seems that the proposed solution is a viable one.

As future work, we plan: (1) to conduct a deeper evaluation of the proposed solutions, also by using real data, to better understand its potential, and to identify possible heuristics refinements and extensions, (2) to enhance the current scheduler to support the space sharing policy, and several forms of job preemption.

8. 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).

9. REFERENCES

[1] S. Baruah, S. Funk, and J. Goossens. Robustness results concerning edf scheduling upon uniform multiprocessors. *IEEE Transactions on Computers*, 52(9):1185–1195, September 2003.

[2] A. Bayucan, R. L. Henderson, L. T. Jasinskyj, C. Lesiak, B. Mann, T. Proett, and D. Tweten. Portable batch system, administrator guide. Technical Report Release: 2.2, MRJ Technology Solutions, Mountain View, CA, USA, November 1999.

[3] S.-H. Chiang, A. C. Arpaci-Dusseau, and M. K. Vernon. 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*, pages 103–127, London, UK, Lect. Notes Comput. Sci. vol. 2537, 2002. Springer-Verlag.

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

[5] H. El-Rewini, T. G. Lewis, and H. H. ALI. *Task Scheduling in Parallel and Distributed Systems*. PTR Prentice Hall, Englewood Cliffs, New Jersey, 1994.

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

[7] D. D. Feitelson, L. Rudolph, and U. Schwiegelshohn. Parallel job scheduling, a status report. In *Job Scheduling Strategies for Parallel Processing 10th International Workshop*, pages 1–16, London, UK, Lect. Notes Comput. Sci. vol. 3277, 2004. Springer-Verlag.

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

[9] A. Fiat and G. J. Woeginger. Online algorithms, the state of the art. In *Developments from a June 1996 seminar on Online algorithms*, London, UK, Lect. Notes Comput. Sci. vol. 1442, 1998. Springer-Verlag.

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

[11] D. B. Jackson, Q. Snell, and M. J. Clement. Core algorithms of the maui scheduler. In *JSSPP '01: Revised Papers from the 7th International Workshop on Job Scheduling Strategies for Parallel Processing*, pages 87–102, London, UK, Lect. Notes Comput. Sci. vol. 2221, 2001. Springer-Verlag.

[12] T. Kawaguchi and S. Kyan. Worst case bound of an lrf schedule for the mean weighted flow-time problem. *SIAM J. Comput.*, 15(4):1119–1129, 1986.

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

[14] S. Martello and P. Toth. *Knapsack Problems, Algorithms and Computer Implementations*. John Wiley and Sons, New York, 1990.

[15] S. Microsystems. Sun one grid engine administration and user guide. Technical Report Part No. 816-2077-12, Sun Microsystems, Inc., Santa Clara, CA, USA, October 2002.

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

[17] D. Puppin, M. Stephenson, W. Lee, and S. Amarasinghe. Convergent scheduling. *The Journal of Instruction Level Parallelism*, 6(1):1–23, September 2004.

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

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

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

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

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

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