Performance and Fairness for Users in Parallel Job Scheduling

Dalibor Klusáček[1,2] and Hana Rudová[1]

[1] Faculty of Informatics, Masaryk University
Botanická 68a, Brno, Czech Republic
[2] CESNET z.s.p.o., Zikova 4, Prague, Czech Republic
{xklusac,hanka}@fi.muni.cz

Abstract. In this work we analyze the performance of scheduling algorithms with respect to fairness. Existing works frequently consider fairness as a job related issue. In our work we analyze fairness with respect to different users of the system as this is a very important real-life problem. First, we discuss how fair are selected popular scheduling algorithms with respect to different users of the system. Next, we present an extension to the well known Conservative backfilling algorithm. Instead of "ad hoc" decisions, the schedule is now created subject to evaluation and optimization. Notably, the fairness is considered as an important metric, which accompanies standard performance related metrics such as slowdown or wait time. To achieve that, an inclusion of fairness as an optimization criterion is proposed. The new extension improves the performance and fairness of Conservative backfilling with respect to other classical techniques such as FCFS, EASY backfilling or aggressive backfilling without reservations.

Key words: Scheduling, Fairness, Metaheuristic, Backfilling

1 Introduction

This paper is inspired by the lessons learned over the past years when analyzing the job scheduling problem in the Czech National Grid Infrastructure MetaCentrum [24]. During that time it became apparent that for satisfactory scheduling several major principles should be met. First of all, the scheduler must guarantee good performance regarding classical performance related metrics such as low job wait times and slowdowns and high system utilization. At the same time, fairness has shown to be one of the most important factors to keep users satisfied. Therefore the users should be treated in a fair fashion, such that the available computing power is fairly distributed among them [13]. Last but not least, the predictability, i.e., planning functionality [10, 25] was found to be very useful as it allows users to better understand when and where their jobs will be executed. In fact, even experienced users often do not understand the scheduling decisions as delivered by existing scheduler that does not use planning functionality.

In order to deal with this situation we have proposed an optimization procedure designed to improve the performance of well known *Conservative backfilling*

algorithm [4, 31, 22]. The choice of Conservative backfilling is straightforward as it allows predictability by establishing reservation for every waiting job. In Conservative backfilling, the plan of job reservations is created in an "ad hoc" fashion as new jobs arrive. This approach may not guarantee good solutions as previously established scheduling decisions are fixed and do not change even when it is obvious that better solution exists. At such situation it is often useful to "reconsider" previous decisions. For this purpose we apply two core strategies: the evaluation procedure and the optimization procedure. The evaluation is used to identify inefficient scheduling decisions and it guides the optimization procedure toward better schedules. Beside common performance related criteria it also focuses on the problem of maintaining *fairness among different users* of the system as this is in fact one of the most important features that a production system should guarantee. Together, the proposed extension improves the performance and fairness of the original Conservative backfilling with respect to other classical techniques such as FCFS, EASY backfilling or aggressive backfilling without reservations.

The paper is organized as follows. First, we define the studied problem and discuss suitable optimization criteria that are lately applied in our study. Next we discuss popular scheduling algorithms, closely describing their strengths and weaknesses while emphasizing fairness related issues. Section 4 presents applied optimization of the Conservative backfilling, i.e., the evaluation procedure and the optimization metaheuristic. Following section presents experimental evaluation where the proposed extension of Conservative backfilling is experimentally compared with the original Conservative backfilling as well as with other popular scheduling algorithms such as FCFS, EASY backfilling or aggressive backfilling without reservations. Finally we conclude our paper with a short summary and we discuss the future work.

2 Problem Description

Let us briefly define the considered job scheduling problem as well as the applied optimization criteria that have been used to express the general requirements on the proposed job scheduler.

2.1 System Description

We consider a classical scenario where the system is managed by one centralized scheduler, which has complete control over all jobs and system resources.

Job represents a user's application that may require one (sequential) or more CPUs (parallel). Also the arrival time and the job processing time are specified. There are no precedence constraints among jobs and we consider neither job preemptions nor migrations from one machine to another. Each job has its owner. When needed, the runtime estimates are precise (perfect) in this study.

The system is composed of one or more computer clusters and each cluster is composed of several machines. So far, we expect that all machines within one

cluster have the same parameters. Those are the number of CPUs per machine and the CPU speed. All machines within a cluster use the Space Slicing processor allocation policy [6], which allows the parallel execution of several jobs at the cluster when the total amount of requested CPUs is less or equal to the number of CPUs of the cluster. Therefore, several machines within the same cluster can be co-allocated to process a parallel job. On the other hand, machines belonging to different clusters cannot be co-allocated to execute the same parallel job.

As we already briefly mentioned in Introduction, the proposed scheduler should guarantee both good performance and fairness. Therefore we now define several criteria that were considered when designing the new scheduler and lately used when evaluating its performance with respect to other existing scheduling techniques.

2.2 Classical Performance Related Criteria

There are several popular metrics used to measure the efficiency of scheduling algorithms. Frequently, makespan and machine usage are used as a general indicator of algorithm's suitability [40, 39]. However, these criteria are not very suitable for systems that are running for a long time. In fact, when the considered time period is long enough, then different algorithms generate similar values of makespan and machine usage. This is not a surprising fact [8, 20] because in such case, the resulting makespan — which is then used to calculate the machine usage — is not controllable by the scheduler since it can never be smaller than the arrival time of the last job plus its processing time. Then, the utilization is rather a function of user activity than of scheduler's performance [8].

Therefore, we have decided to use classical performance related metrics: the avg. response time [6], the avg. wait time [3] and the avg. bounded slowdown [6]. The avg. response time represents the average time a job spends in the system, i.e., the time from its submission to its termination. The avg. wait time is the mean time that the jobs spend waiting before their execution starts. The avg. bounded slowdown is the mean value of all jobs' bounded slowdowns. Slowdown is the ratio of the actual response time of the job to the response time if executed without any waiting. If a job has a very small runtime it often means that the job ended prematurely due to some error. As a result, its slowdown can be huge, which may seriously skew the final mean value. Therefore, so called *bounded slowdown* is often applied [4, 6], where the minimal job runtime is guaranteed to be greater than some predefined time constant, sometimes called a "threshold of interactivity" [6]. However, there is no general agreement concerning the actual value of this threshold. Sometimes it is equal to 10 seconds [4, 6], while different authors use, e.g., 1 minute [37]. Since the resulting value is very sensitive to the applied threshold value, we set the threshold equal to 1 second in this paper. It allows us to eliminate problems related to extremely short jobs while keeping the resulting values close (comparable) to the values of "normal" slowdown in most cases.

All three criteria are to be *minimized*. As pointed out by Feitelson et al. [6], the use of response time places more weight on long jobs and basically ignores

if a short job waits few minutes, so it may not reflect users' notion of responsiveness. Slowdown reflects this situation, measuring the responsiveness of the system with respect to the job length, i.e., jobs are completed within the time proportional to the job length. Wait time criterion supplies the slowdown and response time. Short wait times prevent the users from feeling that the scheduler "forgot about their jobs".

2.3 Fairness Related Criteria

So far, all criteria focused either on the system or the job performance. Still, good performance is not the only aspect that makes the scheduler acceptable. The scheduler must be also fair, i.e., it must guarantee that the available computing power will be fairly distributed among the users of the system. As far as we know there is no widely accepted standardized metric to measure fairness and different authors use different metrics [29, 30, 28, 37, 21, 31]. A *fair start time* (FST) metric is used in [29, 21]. It measures the influence of later arriving jobs on the execution start time of currently waiting jobs. FST is calculated for each job, by creating a schedule assuming no later jobs arrive. The resulting "unfairness" is the difference between FST and the actual start time. Similar metric is so called *fair slowdown* [31]. The fair slowdown is computed using FST and can be used to quantify the fairness of a scheduler by looking at the percentage of jobs that have a higher slowdown than their fair slowdown [31]. Another metric measures to what extent each job was able to receive its "share" of the resources [30, 28]. The basic idea is that each job "deserves" $1/n^t$ of the resources, where n^t is the number of jobs present in the system at the given time t. The "unfairness" is computed by comparing the resources consumed by a job with the resources deserved by the job. An overview of existing techniques including discussion of their suitability can be found in [26].

Fairness is usually understood and represented as a *job related* metric, meaning that every job should be served in a fair fashion with respect to other jobs [29, 30, 28, 37, 21, 31]. Such requirements are already partially covered by the common performance criteria like the slowdown and the wait time that were both discussed earlier. Moreover, job fairness has been also reflected in the design of common scheduling algorithms (see Section 3) and the results concerning selected job fairness indicators are well known [30, 28, 37, 31]. In this work, we aim to guarantee fair performance to *different users* of the system as well. Therefore, we apply a well know fair-share principle [13] with respect to different users of the system. Fair-share tries to minimize the differences among the normalized mean wait times of all users. Let o be a given user (job owner) in the system and $\mathcal{JOBS}_o$ be the set containing jobs of user o. Let r_j and S_j denote the arrival time and start time of given job j respectively. Then the *normalized user wait time* ($NUWT_o$) for each user (job owner) o is calculated as shown by Formula 1.

$$NUWT_o = \frac{TUWT_o}{TUSA_o} \tag{1}$$

$$TUWT_o = \sum_{j \in \mathcal{JOBS}_o} (S_j - r_j) \tag{2}$$

$$TUSA_o = \sum_{j \in \mathcal{JOBS}_o} (p_j \cdot usage_j) \tag{3}$$

Normalized user wait time $NUWT_o$ is the *total user wait time* (see $TUWT_o$ in Formula 2) divided (normalized) by the so called *total user squashed area* (see $TUSA_o$ in Formula 3), which can be described as the sum of products of the job runtime (p_j) and the number of requested processors ($usage_j$). This user-oriented metric is based on more general *total squashed area* metric proposed in [3]. The normalization is used to prioritize less active users over those who utilize the system resources very frequently [13]. Such normalization is commonly used as can be seen, e.g., in the Czech National Grid Infrastructure MetaCentrum [24] where the normalization is used when monitoring and adjusting fairness in the production TORQUE scheduling system [1] as well as it was used earlier in the PBS Pro scheduling system [11]. Similar approach has been also adopted in the ASCI Blue Mountain supercomputer cluster when establishing job priorities in the fair-share mechanism [13].

The normalized user wait time ($NUWT_o$) can be used "on the fly" by the scheduling algorithm to dynamically prioritize the users. Also, it can be used in the graphs with the experimental results (e.g., Fig. 2) to reflect the resulting fairness of the applied scheduling algorithm. In this case, the interpretation is following. The closer the resulting $NUWT_o$ values of all users are to each other, the higher is the fairness. If the $NUWT_o$ value is less than 1.0, it means that the user spent more time by computing than by waiting, which is advantageous. Similarly, values greater than 1.0 indicate that the total user wait time is larger than the computational time of his or hers jobs.

3 Fairness vs. Performance in Scheduling Algorithms

In this section we recapitulate several popular scheduling algorithms that are widely used both in practice and in the literature. We will discuss their strengths and weaknesses with respect to classical performance related metrics as well as with respect to fairness related issues.

All production systems support trivial *First Come First Served (FCFS)* scheduling policy [11, 23]. FCFS always schedules the first job in the queue, checking the availability of the resources required by such job. If all the resources required by the first job in the queue are available, it is immediately scheduled for the execution, otherwise FCFS waits until all required resources become available. While the first job is waiting for the execution none of the remaining jobs can be scheduled, even if the required resources are available. Despite its simplicity, FCFS approach presents several advantages. It does not require an estimated processing time of the job and it guarantees that the response time of a job that arrived earlier does not depend on the execution times of jobs that arrived later. As there is no "queue jumping" FCFS is in some sense a very fair

scheduler [25, 31, 30]. On the other hand, if parallel jobs are scheduled then this fairness related property often implies a low utilization of the system resources, that cannot be used by some "less demanding" job(s) from the queue [29, 23]. To solve this problem algorithms based on backfilling are frequently used [25].

Algorithms using *backfilling* are an optimization of the FCFS algorithm that try to maximize the resource utilization [23]. There are several variants of backfilling algorithms. The most popular one is the aggressive *EASY backfilling* [25]. It works as FCFS but when the first job in the queue cannot be scheduled immediately EASY backfilling calculates the earliest possible starting time for the first job using the processing time estimates of running jobs. Then, it makes a reservation to run the job at this pre-computed time. Next, it scans the queue of waiting jobs and schedules immediately every job not interfering with the reservation of the first job [23]. This helps to increase the resource utilization, since idle resources are *backfilled* with suitable jobs, while decreasing the average job wait time.

EASY Backfilling takes an aggressive approach that allows short jobs to skip ahead provided they do not delay the job at the head of the queue. The price for improved utilization of EASY Backfilling is that execution guarantees cannot be made because it is hard to predict the size of delays of jobs in the queue. Since only the first job gets a reservation, the delays of other queued jobs may be, in general, unbounded [25][3]. Therefore, without further control, EASY does not guarantee good fairness.

In order to prevent such situation, the number of reservations can be increased. In case of slack-based [34] and selective backfilling [31] the number of jobs with a reservation is related to their current wait time and slowdown respectively. *Conservative backfilling* [4, 31, 22] makes reservation for every queued job which cannot be executed at the given moment. It means that backfilling is performed only when it does not delay any previous job in the queue. Clearly, this reduce the core problem of EASY backfilling where jobs close to but not yet at the head of the queue can be significantly delayed. The price paid is that the number of jobs that can utilize existing gaps[4] is reduced, implying that more gaps are left unused in Conservative backfilling than in EASY backfilling [26]. Still, both approaches lead to significant performance improvements compared to FCFS [26, 7]. As the scheduling decisions are made upon job submittal, it can be predicted when each job will run, giving the users execution guarantees. Users can then plan ahead based on these guaranteed response times. Obviously, there is no danger of starvation as a reservation is made for every job that cannot be executed immediately. Apparently, such approach places a greater emphasis on predictability [25, 4] and it is a good compromise between "fair" but inefficient FCFS and "unfair" but efficient EASY backfilling.

[3] If a job is not the first in the queue, new jobs that arrive later may skip it in the queue. While such jobs do not delay the first job in the queue, they may delay all other jobs and the system cannot predict when a queued job will eventually run [25].

[4] Some authors [26] call the unused CPU time slot a "hole" while others [25, 36] prefer the term "gap".

All previously mentioned variants of backfilling require that each job specifies its estimated execution time. Therefore, existing systems also support backfilling without reservations [19, 11] where estimates are not needed at all. In order to maintain fairness among users, the queue(s) can be ordered according to some priority mechanism such as fair-share. For example, the TORQUE scheduler [1] currently used in MetaCentrum [24] uses backfilling without reservations where each queue is ordered according to priorities computed using the fair-share principle. Here, the users of the system are prioritized according to the fair-share mechanism that balances the amount of consumed CPU time among users. This means that all jobs belonging to a given user get the priority that is equal to the user's priority[5]. Still, as in the case of EASY backfilling, such techniques cannot guarantee starvation-free behavior and additional mechanisms are needed to minimize the risk that the delays of queued jobs become very large.

In this section we tried to mention the most popular scheduling algorithms and we tried to demonstrate their pros and cons. None of these algorithms suits our needs perfectly. FCFS is somehow fair but inefficient. EASY backfilling improves the performance significantly but at some situations may degrade the performance for "unlucky" jobs. Similarly, backfilling without reservations do not need estimates but it cannot guarantee starvation-free behavior.

From this point of view, Conservative backfilling is a good candidate for further extension. First, as each job gets a reservation waiting jobs cannot be delayed by lately arriving jobs, which is fair, at least from the user's point of view. Second, job reservations, i.e., the plan of execution, are good for the users as they can get some sort of guarantee and they "know what is happening". Last but not least, the prepared plan of execution can be easily evaluated with respect to selected optimization criteria, covering both performance and fairness related objectives. Therefore, possible inefficiencies that can appear in classical Conservative backfilling can be identified and fixed. For this purpose some form of metaheuristic algorithm seems to be a natural solution. In the next section we describe such an extension of the Conservative backfilling.

4 Optimization of Conservative Backfilling

As we mentioned in previous text, we found Conservative backfilling to be a good initial scheduling technique for our purposes, which included requests of good performance, fair distribution of available computing power among users and predictability. In this section we describe the two fundamental techniques used to improve the overall performance of Conservative backfilling. Those techniques are *evaluation* of existing solution and an *optimization metaheuristic* that improves the quality of generated solution with respect to considered criteria. We start with a description of the evaluation method.

[5] The priority is computed using the Formula 1 that represents the normalized user wait time $NUWT_o$.

4.1 Evaluation Procedure

The purpose of the evaluation procedure is to compare two different schedules and decide, which one is better with respect to applied optimization criteria. As we already discussed in Section 2, we focus both on the classical and the fairness related criteria. We use the avg. wait time (WT), the avg. response time (RT) and the avg. bounded slowdown (BSD) to measure the performance of the scheduling algorithm. Each such metric can be easily used when deciding, which solution is better — the one having smaller values of given metric. When considering fairness with respect to different users the situation is more complicated. The fairness related *normalized user wait time* ($NUWT_o$) described in Section 2.3 cannot be directly used as it is only a per user metric. For our purpose we needed a function that for given schedule returns a *single value*. Therefore, we have proposed a criterion called *fairness* (F), which is computed as shown by Formula 5.

$$UWT = \frac{1}{u} \sum_{o=1}^{u} NUWT_o \tag{4}$$

$$F = \sum_{o=1}^{u} \left(UWT - NUWT_o \right)^2 \tag{5}$$

First, we calculate the mean *user wait time* (UWT) using the values of $NUWT_o$ as shown in Formula 4. Then the fairness F is calculated by the Formula 5. The squares in F definition guarantee that only positive numbers are summed and that higher deviations from the mean value are more penalized than the small ones. This approach has been inspired by the widely used *Least Squares method* [38] where similar formula of squared residuals is minimized when fitting values provided by a model to observed values.

The fairness (F) criterion is used during the evaluation of performed scheduling decisions, i.e., "inside" the optimization procedure (see Section 4.2). When two possible solutions are available, then the values of F are computed for both of them. The one having smaller F value is considered as more fair. The squares used during computation of F help to highlight unfair assignments, which is very favorable when performing scheduling decisions. Sadly, the squares basically prevent us to reasonably interpret the resulting F values as it was possible in case of the $NUWT_o$ criterion (see discussion in Section 2.3). This is the reason why $NUWT_o$ is used in graphs to display how fair the solution has been with respect to different users while F is used when evaluating two different schedules "inside" the optimization algorithm.

Together, there are four criteria to be optimized simultaneously. Each criterion produces one value characterizing the solution. The final decision on which of the two solutions is better is implemented in separate function, called SELECTBETTER($schedule_A, schedule_B$) which is shown in Algorithm 1. It is a form of a *weight function*, which is often used when solving multi-criteria optimization problems [39, 18, 17]. The SELECTBETTER function is an extended version of the

function that has been already successfully used in our previous works [17, 18]. This extension includes the fairness criterion.

Algorithm 1 SELECTBETTER($schedule_A, schedule_B$)

```
1: compute BSD_A, WT_A, RT_A, F_A according to schedule_A;
2: compute BSD_B, WT_B, RT_B, F_B according to schedule_B;
3: v_BSD := (BSD_A − BSD_B)/BSD_A;
4: v_WT := (WT_A − WT_B)/WT_A;
5: v_RT := (RT_A − RT_B)/RT_A;
6: v_F := (F_A − F_B)/F_A;
7: weight := v_BSD + v_WT + v_RT + v_F;
8: if weight > 0 then
9:    return schedule_B;
10: else
11:    return schedule_A;
12: end if
```

This function uses two inputs — the two solutions that will be compared. The $schedule_A$ may represent existing (previously accepted) solution while $schedule_B$ represents the newly created *candidate solution*, a product of optimization. First, the values of used objective functions are computed for both schedules (lines 1–2). Using them, decision variables v_{BSD}, ..., v_F are computed (see lines 3–6). Their meaning is following: when the decision variable is positive it means that the $schedule_B$ is better than the $schedule_A$ with respect to the applied criterion. Strictly speaking, decision variable defines percentual improvement or deterioration in the value of objective function of $schedule_B$ with respect to the $schedule_A$. Some trivial correction is needed when the denominator is equal to zero, to prevent division by zero error. To keep the code clear we do not present it here. It can easily happen, that for given $schedule_B$ some variables are positive while others are negative. In our implementation the final decision is taken upon the value of the *weight* (line 7), which is computed as the (weighted) sum of decision variables. If desirable, the "importance" of each decision variable can be adjusted using a predefined weight constant. However, proper selection of these weights is not an easy task. In this particular case, all decision variables are considered as equally important and no additional weight constants are used. When the resulting *weight* is positive the (candidate) $schedule_B$ is returned as a better schedule. Otherwise, the (existing) $schedule_A$ is returned.

4.2 Optimization Algorithm

Newly arriving jobs are added into the schedule using classical Conservative backfilling algorithm. It means that the earliest suitable time slot is found in existing schedule. Such initial schedule ($schedule_{initial}$) is periodically optimized with an optimization algorithm. It is a simple metaheuristic that tries to optimize

the existing $schedule_{initial}$. The algorithm includes an important feature that is typical for the *Tabu search*-based algorithms [9, 27]. It is a short term memory called *tabu list* where few previously manipulated jobs are stored. If the algorithm is trying to move some job then this change is not allowed in case that this job is present in the tabu list. It has limited size and the oldest item is always removed when the list becomes full. Tabu list helps to protect the algorithm against short cycles where the same few jobs are repeatedly selected as the move candidates. The main structure of the algorithm is based on a procedure that has been already successfully used in our previous work which focused on a different problem involving minimizing the number of late jobs [17].

Algorithm 2 TabuSearch($schedule_{initial}, iterations, time\_limit$)

```
1: schedule_new := schedule_initial; schedule_best := schedule_initial; tabu_list := ∅;
2: i := 0;
3: while (i < iterations and time_limit not exceeded) do
4:    i := i + 1;
5:    job := select random job from schedule_new such that job ∉ Tabu;
6:    if job = null then
7:       tabu_list := ∅;    (all jobs were tested, reset the tabu list)
8:       continue;
9:    end if
10:   remove job from schedule_new;
11:   compress schedule_new;
12:   move job into earliest suitable gap in schedule_new;
13:   schedule_best := SelectBetter(schedule_best, schedule_new);
14:   schedule_new := schedule_best;    (update/reset candidate schedule)
15:   if tabu_list is full then
16:      remove the oldest item;
17:   end if
18:   tabu_list := tabu_list ∪ job;
19: end while
20: return schedule_best;
```

TabuSearch($schedule_{initial}, iterations, time\_limit$) optimization algorithm is described in Algorithm 2. It has three inputs — the schedule that will be optimized, the maximal number of iterations and a time limit. In each iteration, one random non-tabu job selected (line 5). Once the job is selected, it is removed from its current position and the schedule is immediately compressed. The compression is designed to shift reservations to earlier time slots that could have appeared as a result of the job removal [14]. This procedure is an analogy to the method used in Conservative backfilling when job terminates earlier due to an overestimated runtime estimate [25]. During the compression the relative ordering of job start times is kept [14]. Next, the removed job is returned to the compressed schedule into the earliest suitable gap and this

new schedule is evaluated with respect to applied criteria in the SELECTBETTER($schedule_{best}, schedule_{new}$) function (see Algorithm 1). If this attempt is successful SELECTBETTER returns $schedule_{new}$ as the new $schedule_{best}$, otherwise $schedule_{best}$ is not changed (line 13). Next, the $schedule_{new}$ is updated with the $schedule_{best}$ (line 14). Finally, the job is placed into the *tabu_list* (line 18) — so that it cannot be chosen in the next few iterations — and a new iteration of the Tabu search starts. If in some iteration all jobs are already in the tabu list, then the list is emptied and another iteration starts (line 7)[6]. The cycle continues until the predefined number of iterations or the given time limit is reached (lines 3). Then, the $schedule_{best}$ is returned as the newly found solution (line 20).

When applied, TABUSEARCH is executed every 5 minutes of simulation time. Here we were inspired by the actual setup of the scheduler [1] used in the MetaCentrum, which performs periodic priority updates of jobs waiting in the queues with a similar frequency. The maximum number of iterations is equal to the number of currently waiting jobs (schedule size) multiplied by 2. The *time_limit* variable was set to be 2 seconds, which is usually enough to try all iterations. However, when some higher priority event such as new job arrival or job completion is detected during TABUSEARCH execution, the *time_limit* is immediately set to 0 and the TABUSEARCH terminates. Therefore, the optimization phase cannot cause any significant delays concerning job processing and the potential overhead of optimization is practically eliminated [17].

5 Experiments

This section presents the experimental evaluation where the proposed Tabu Search optimization technique is experimentally compared with selected popular scheduling algorithms. Beside common performance related criteria also the fairness related issues of considered scheduling algorithms are analyzed here.

5.1 Experimental Setup

All experiments have been computed on an Intel Xeon 3.0 GHz machine with 12 GB of RAM using the GridSim [33] based *Alea* simulator we have implemented [15].

The proposed Tabu search-based optimization (TS) of Conservative backfilling has been evaluated against several existing algorithms that have been all closely discussed in Section 3. We have considered FCFS, aggressive backfilling without reservations (BF), EASY backfilling (BF-EASY), Conservative backfilling (BF-CONS) and the aggressive backfilling without reservations prioritized according to fair-share (BF-FAIR).

[6] In the current implementation the tabu list has maximum size of 10 jobs. If all jobs from the current schedule are in the *tabu_list*, it means that there are at most 10 jobs in the whole schedule.

Six different data sets from the Parallel Workloads Archive [5] have been used in the simulation: MetaCentrum (806 CPUs, 103,656 jobs during 5 months), SDSC BLUE (1,152 CPUs, 243,314 jobs during 34 months), CTC SP2 (338 CPUs, 77,222 jobs during 11 months), HPC2N (240 CPUs, 202,876 jobs during 42 months), SDSC SP2 (128 CPUs, 59,725 jobs during 24 months) and KTH SP2 (100 CPUs, 28,489 jobs during 11 months). If available, the recommended "cleaned" versions of workload logs are always used. Detailed descriptions of these logs are available in the Parallel Workloads Archive [5].

In the experiments, the avg. response time, the avg. wait time and the avg. bounded slowdown have been measured as the standard performance related metrics. A bubble chart is used to display corresponding values of wait time and slowdown simultaneously — the y-axis depicts the avg. wait time while the size of the circle represents the avg. bounded slowdown. The actual bounded slowdown value is shown as a label bellow each circle (see Fig. 1).

Concerning fairness, the resulting normalized user wait times $NUWT_o$ were collected for all users. In the next step, we have removed all $NUWT_o$ values that belonged to users who submitted only one job as this represents an extreme situation. When such user submits the first (and only) job into the system, the job gets some default priority, since the system cannot compute $NUWT_o$ that is normally used to establish the job priority[7]. At the same time, other jobs in the queue often have higher priority, therefore this single job may be delayed by other high priority jobs. However, as the user do not submit any other job, there is no way for the scheduler to "fix" this unfair assignment and the resulting $NUWT_o$ may be quite high. Once the set of $NUWT_o$ values was reduced as explained above, the $NUWT_o$ values were interpreted in two different ways. The cumulative distribution function (CDF) of users' normalized wait times presents how different scheduling algorithms affect the resulting $NUWT_o$ values for all users of the system. In this case, the CDF is a $f(x)$-like function showing the probability that the $NUWT_o$ of given user o is less than or equal to x. In another words, the CDF represents percentage of users having their $NUWT_o$ less than or equal to x. The steeper is the resulting curve the closer (i.e., more fair) were the $NUWT_o$ values of different users. As the resulting distributions often have very long tails, the maximal $NUWT_o$ shown on the x-axis is bounded by 10.0 for better visibility. The CDFs are accompanied with two additional metrics which show the arithmetic mean of all normalized user wait times and the corresponding standard deviation. The smaller the mean and the standard deviation are the lower were the $NUWT_o$ values and the closer (i.e., more fair) they were. When two or more algorithms have similar CDFs, these two metrics helps to highlight the differences among algorithms as they can highlight the influence of the distribution's long tail.

[7] $NUWT_o$ cannot be computed because both $TUWT_o$ and $TUSA_o$ are not known unless at least one job of given user o completes (see Formulas 1–3).

5.2 Experimental Results and Discussion

Fig. 1 presents the avg. wait time, the avg. bounded slowdown (1st and 3rd row) and the avg. response time (2nd and 4th row) for all six data sets. The fairness related results for all data sets are shown in Fig. 2. As discussed in Section 5.1, two different graphs are used to capture the resulting normalized user wait times ($NUWT_o$). The CDFs of $NUWT_o$ distributions are shown in the first and third row in Fig. 2. The bar charts with the mean of all normalized user wait times and the corresponding standard deviations are shown in the second and fourth row in Fig. 2.

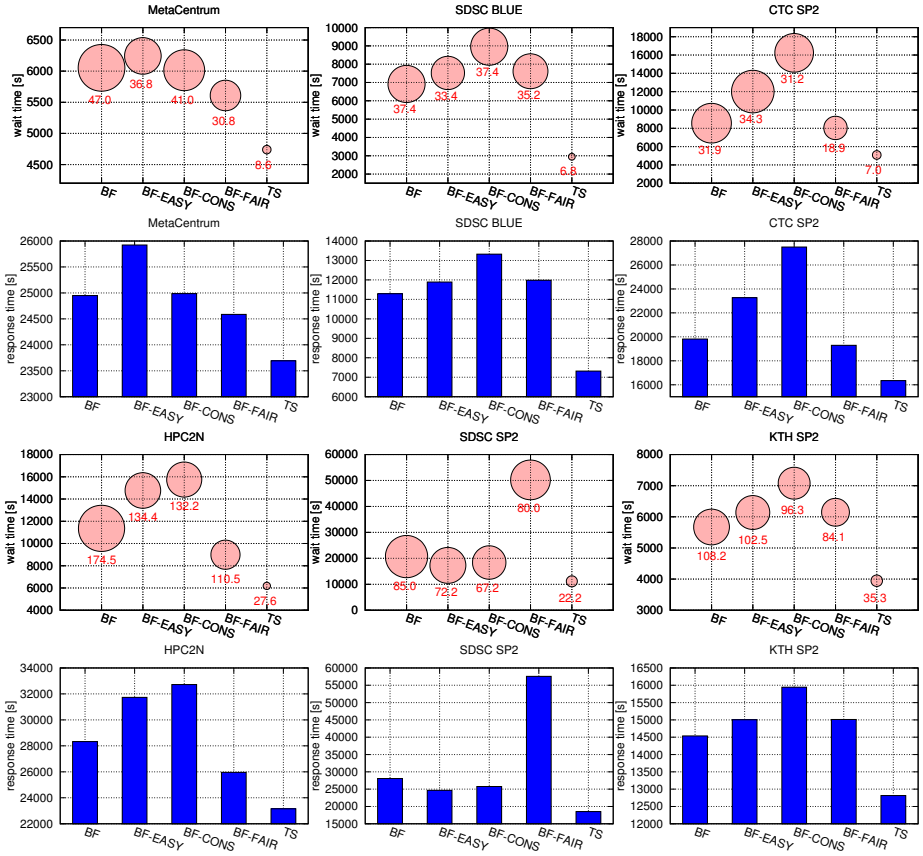

Fig. 1. The avg. wait time and the avg. slowdown (1st and 3rd row), and the avg. response time (2nd and 4th row) for all six data sets.

Let us discuss the results of the experiments. In all experiments FCFS performed very bad, which is not surprising as the applied workloads represent reasonably utilized systems with parallel jobs where FCFS is known to be inef-

ficient [12, 23, 25]. Therefore — with the exception made in case of CDFs — the results of FCFS are not presented in the charts for better visibility, as in all cases the results of FCFS were very bad and off-scale high.

Concerning the avg. wait time (1st and 3rd row in Fig. 1) and the avg. response time (2nd and 4th row in Fig. 1), original Conservative backfilling (BF-CONS) does not work very well with respect to other backfilling algorithms, as both BF-EASY or BF produce better results in most cases. This is not surprising as these issues have been already addressed in several works [31, 32, 26]. Basically, the problem here is that establishing reservation for every job can be less efficient than aggressive approaches as used in BF or BF-EASY. Reservations decrease the opportunities for backfilling, due to the blocking effect of the reserved jobs in the schedule [31, 4]. On the other hand, the slowdown (see circle labels in the bubble charts in Fig. 1) is often slightly better for BF-CONS as no job can be delayed. This is a normal behavior also observed in previous works [31, 4]. BF-FAIR is also competitive, however in case of SDSC SP2 it does not perform very well with respect to the avg. wait time. On the other hand, TS produces the best wait times, response times and slowdowns in all six cases. Clearly, TS significantly improves the otherwise relatively weak performance of Conservative backfilling. These results indicate that the "ad hoc" manner used to establish reservations in BF-CONS is not very efficient and can be easily improved using the evaluation and optimization techniques.

In case of fairness related criteria (see Fig. 2), the results clearly demonstrate that standard solutions such as FCFS, BF, BF-EASY or BF-CONS are not very good to guarantee good fairness with respect to different users since they do not involve any suitable mechanism for this purpose. Especially FCFS is truly unfair for users as can be seen in the CDFs (1st and 3rd row in Fig. 2). In general, BF, BF-EASY and BF-CONS produce worse results than BF-FAIR or TS in most cases. While the differences in the CDFs are not huge, there are typically several users with very high (unfair) $NUWT_o$ when BF, BF-EASY or BF-CONS is used, respectively. This unfairness significantly increases the arithmetic mean of all normalized user wait times and the corresponding standard deviation as can be seen in the second and fourth row in Fig. 2.

From this point of view, a simple extension involving fair-share based priorities as applied in BF-FAIR algorithm can significantly improve the fairness of the solution with respect to different users. In most situations BF-FAIR generates better, i.e., steeper CDFs of normalized user wait times (see 1st and 3rd row in Fig. 2) as well as better, i.e., lower mean values and standard deviations (2nd and 4th row in Fig. 2) than FCFS or other backfilling algorithms. Again, TS optimization procedure shows great potential when improving the fairness of the original BF-CONS. TS can guarantee fairness on the same or even higher level as BF-FAIR algorithm does, thanks to the applied extensions that involve schedule evaluation and optimization. Clearly, good results in standard performance metrics (see Fig. 1) are not achieved at the expenses of fairness (see Fig. 2).

As discussed in Section 2.1, if needed, the runtime estimates are precise (perfect) in this study. This setup has been used in order to provide "ideal" conditions

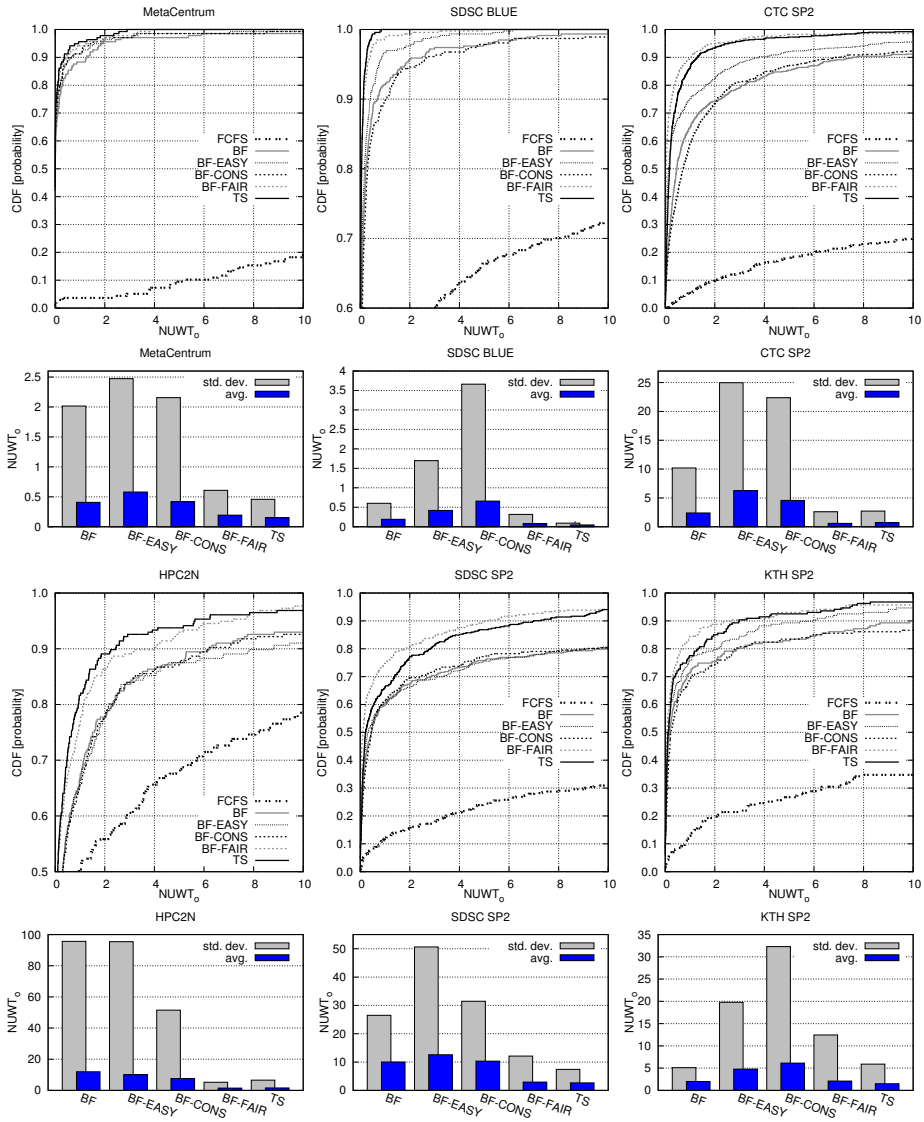

Fig. 2. The CDFs of normalized user wait times (1st and 3rd row), and the mean of normalized user wait times and the corresponding standard deviation (2nd and 4th row) for all six data sets.

for all algorithms that somehow use reservation(s). This approach minimizes the effect of inaccuracy that may sometimes produce "confusing" results [35, 36]. One may suggest that the proposed extension of Conservative backfilling may not work that well as soon as realistic, i.e., inaccurate estimates are used. However, as we observed in our recent studies [14, 16], the inaccuracy can be handled

efficiently if proper "recovery" methods are applied. For example, the inefficiency caused by early job completions can be minimized by applying schedule compression as discussed in Section 4.2 or in [25, 14]. Moreover, as the optimization procedure follows a "gap filling" approach (see line 12 in Algorithm 2) it can be flexibly used to further improve the schedule once the compression phase terminates [14, 16]. Similarly to, e.g., EASY backfilling, it is favorable when the users' estimates are heterogeneous. When there are only few popular estimates (e.g., 3 popular estimates), the performance of the proposed technique is similar to the performance of the backfilling solutions [14]. At such situation, the limited diversity of runtime estimates prevents us from building efficient schedules, since there is no good opportunity for successful optimization [14].

6 Conclusion

This paper addressed a real life-based job scheduling problem. The goals were to maintain the fairness among different users of the system while keeping good performance regarding classical criteria such as slowdown or wait time. Several existing algorithms have been analyzed with respect to these two goals. Moreover, an extension of the well known Conservative backfilling has been proposed in order to guarantee good fairness and performance. The extension uses optimization procedure, which allows to improve the quality of the schedule. Optimization is guided by the evaluation that is performed subject to applied objective functions. Experimental evaluation demonstrates that the proposed extension represents significant improvement by means of fairness and performance over several existing algorithms including FCFS, Conservative and EASY backfilling as well as aggressive backfilling without reservations.

Just like the original Conservative backfilling, the current solution still supports predictability as reservations are established for every job. However, the optimization technique can delay particular jobs with respect to their initial reservations if it improves the overall quality of the schedule. As this behavior may be problematic in some cases, we plan to solve this issue in the future. Currently, we are working on a closely related scheduling approach involving evaluation and optimization algorithms [2] within the production TORQUE scheduler that is used in the Czech National Grid Infrastructure MetaCentrum.

Acknowledgments. We appreciate the gracious support of the Grant Agency of the Czech Republic under the grant No. P202/12/0306. The access to the MetaCentrum computing facilities provided under the programme "Projects of Large Infrastructure for Research, Development, and Innovations" LM2010005 funded by the Ministry of Education, Youth, and Sports of the Czech Republic is highly appreciated. The MetaCentrum workload log was graciously provided by the Czech National Grid Infrastructure MetaCentrum. The CTC SP2, KTH SP2, SDSC SP2 and HPC2N workload logs were graciously provided by Dan Dwyer, Lars Malinowsky, Victor Hazlewood and Ake Sandgren, respectively. The work-

load log from the SDSC Blue Horizon was graciously provided by Travis Earheart and Nancy Wilkins-Diehr.

References

1. Adaptive Computing Enterprises, Inc. *TORQUE Admininstrator Guide, version 3.0.3*, February 2012. `http://www.adaptivecomputing.com/resources/docs/`.
2. Václav Chlumský, Dalibor Klusáček, and Miroslav Ruda. The extension of TORQUE scheduler allowing the use of planning and optimization algorithms in Grids. *Computer Science*, 2012. To appear.
3. Carsten Ernemann, Volker Hamscher, and Ramin Yahyapour. Benefits of global Grid computing for job scheduling. In *GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 374–379. IEEE, 2004.
4. Dror G. Feitelson. Experimental analysis of the root causes of performance evaluation results: A backfilling case study. *IEEE Transactions on Parallel and Distributed Systems*, 16(2):175–182, 2005.
5. Dror G. Feitelson. Parallel workloads archive (PWA), February 2012. `http://www.cs.huji.ac.il/labs/parallel/workload/`.
6. Dror G. Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C. Sevcik, and Parkson Wong. Theory and practice in parallel job scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1291 of *LNCS*, pages 1–34. Springer Verlag, 1997.
7. Dror G. Feitelson and Ahuva M. Weil. Utilization and predictability in scheduling the IBM SP2 with backfilling. In *12th International Parallel Processing Symposium*, pages 542–546. IEEE, 1998.
8. Eitan Frachtenberg and Dror G. Feitelson. Pitfalls in parallel job scheduling evaluation. In Dror G. Feitelson, Eitan Frachtenberg, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 3834 of *LNCS*, pages 257–282. Springer Verlag, 2005.
9. Fred W. Glover and Manuel Laguna. *Tabu search*. Kluwer, 1998.
10. Matthias Hovestadt, Odej Kao, Axel Keller, and Achim Streit. Scheduling in HPC resource management systems: Queuing vs. planning. In *Job Scheduling Strategies for Parallel Processing*, volume 2862 of *LNCS*, pages 1–20. Springer, 2003.
11. James Patton Jones. *PBS Professional 7, administrator guide*. Altair, April 2005.
12. Peter J. Keleher, Dmitry Zotkin, and Dejan Perkovic. Attacking the bottlenecks of backfilling schedulers. *Cluster Computing*, 3(4):245–254, 2000.
13. Stephen D. Kleban and Scott H. Clearwater. Fair share on high performance computing systems: What does fair really mean? In *Third IEEE International Symposium on Cluster Computing and the Grid (CCGrid'03)*, pages 146 – 153. IEEE Computer Society, 2003.
14. Dalibor Klusáček. *Event-based Optimization of Schedules for Grid Jobs*. PhD thesis, Masaryk University, 2011.
15. Dalibor Klusáček and Hana Rudová. Alea 2 – job scheduling simulator. In *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques (SIMUTools 2010)*. ICST, 2010.
16. Dalibor Klusáček and Hana Rudová. Handling inaccurate runtime estimates by event-based optimization. In *Cracow Grid Workshop 2010 Abstracts (CGW'10)*, Cracow, Poland, 2010.

17. Dalibor Klusáček and Hana Rudová. Efficient Grid scheduling through the incremental schedule-based approach. *Computational Intelligence*, 27(1):4–22, 2011.
18. Dalibor Klusáček, Hana Rudová, Ranieri Baraglia, Marco Pasquali, and Gabriele Capannini. Comparison of multi-criteria scheduling techniques. In *Grid Computing Achievements and Prospects*, pages 173–184. Springer, 2008.
19. A LaTorre, J Pena, V Robles, and P De Miguel. Supercomputer scheduling with combined evolutionary techniques. In Fatos Xhafa and Ajith Abraham, editors, *Metaheuristics for Scheduling in Distributed Computing Environments Studies in Computational Intelligence*, volume 146, pages 95–120. Springer, 2008.
20. Cynthia Bailey Lee. *On the User-Scheduler Relationship in High-Performance Computing*. PhD thesis, University of California, San Diego, 2009.
21. Vitus Joseph Leung, Gerald Sabin, and Ponnuswamy Sadayappan. Parallel job scheduling policies to improve fairness: a case study. Technical Report SAND2008-1310, Sandia National Laboratories, 2008.
22. Bo Li and Dongfeng Zhao. Performance impact of advance reservations from the Grid on backfill algorithms. In *Sixth International Conference on Grid and Cooperative Computing (GCC 2007)*, pages 456 –461, 2007.
23. David A. Lifka. The ANL/IBM SP Scheduling System. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 949 of *LNCS*, pages 295–303. Springer-Verlag, 1995.
24. MetaCentrum, February 2012. `http://www.metacentrum.cz/`.
25. Ahuva W. Mu'alem and Dror G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 12(6):529–543, 2001.
26. John Ngubiri. *Techniques and Evaluation of Processor Co-allocation in Multicluster Systems*. PhD thesis, Radboud University Nijmegen, 2008.
27. Michael Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Prentice Hall, 2002.
28. Gerald Sabin. *Unfairness in parallel job scheduling*. PhD thesis, The Ohio State University, 2006.
29. Gerald Sabin, Garima Kochhar, and P. Sadayappan. Job fairness in non-preemptive job scheduling. In *International Conference on Parallel Processing (ICPP'04)*, pages 186–194. IEEE Computer Society, 2004.
30. Gerald Sabin and P. Sadayappan. Unfairness metrics for space-sharing parallel job schedulers. In Dror G. Feitelson, Eitan Frachtenberg, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 3834 of *LNCS*, pages 238–256. Springer, 2005.
31. Srividya Srinivasan, Rajkumar Kettimuthu, Vijay Subramani, and P. Sadayappan. Selective reservation strategies for backfill job scheduling. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 2537 of *LNCS*, pages 55–71. Springer Verlag, 2002.
32. Srividya Srinivasan, Rajkumar Kettimuthu, Vijay Subrarnani, and P. Sadayappan. Characterization of backfilling strategies for parallel job scheduling. In *Proceedings of 2002 International Workshops on Parallel Processing*, pages 514–519. IEEE Computer Society, 2002.
33. Anthony Sulistio, Uros Cibej, Srikumar Venugopal, Borut Robic, and Rajkumar Buyya. A toolkit for modelling and simulating data Grids: an extension to GridSim. *Concurrency and Computation: Practice & Experience*, 20(13):1591–1609, 2008.
34. David Talby and Dror G. Feitelson. Supporting priorities and improving utilization of the IBM SP scheduler using slack-based backfilling. In *IPPS '99/SPDP '99: Proceedings of the 13th International Symposium on Parallel Processing and*

the 10th Symposium on Parallel and Distributed Processing, pages 513–517. IEEE Computer Society, 1999.
35. Dan Tsafrir. Using inaccurate estimates accurately. In Eitan Frachtenberg and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 6253 of *LNCS*, pages 208–221. Springer Verlag, 2010.
36. Dan Tsafrir and Dror G. Feitelson. The dynamics of backfilling: Solving the mystery of why increased inaccuracy may help. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 131–141. IEEE, 2006.
37. Sangsuree Vasupongayya and Su-Hui Chiang. On job fairness in non-preemptive parallel job scheduling. In S. Q. Zheng, editor, *International Conference on Parallel and Distributed Computing Systems (PDCS 2005)*, pages 100–105. IASTED/ACTA Press, 2005.
38. John Wolberg. *Data Analysis Using the Method of Least Squares: Extracting the Most Information from Experiments*. Springer, 2006.
39. Fatos Xhafa and Ajith Abraham. Computational models and heuristic methods for Grid scheduling problems. *Future Generation Computer Systems*, 26(4):608–621, 2010.
40. Fatos Xhafa, Javier Carretero, Enrique Alba, and Bernabé Dorronsoro. Design and evaluation of Tabu search method for job scheduling in distributed environments. In *International Symposium on Parallel and Distributed Processing (IPDPS 2008)*, pages 1–8. IEEE, 2008.